Proceedings of the Second Stratego Users Day

Eelco Visser (editor)

www.stratego-language.org

Institute of Information and Computing Sciences
Universiteit Utrecht
February 8, 2001

Address:
Institute of Information and Computing Sciences
Universiteit Utrecht
P.O.Box 80089
3508 TB Utrecht
email: visser@acm.org
http://www.cs.uu.nl/~visser/

Preface

These are the proceedings of the Second Stratego Users Day, which was held on February 8, 2001 at Utrecht University. The Users Day was preceded by a full day tutorial on February 7. The workshop and tutorial were supported by the Software Technology Group of the Institute for Information and Computing Sciences.

The workshop was attended by:

- Arne de Bruijn (Utrecht University)
- Eelco Dolstra (Utrecht University)
- Magne Haveraaen (University of Bergen)
- Ronald van Halen (Lucent)
- Karl Trygve Kalleberg (University of Bergen)
- Ralf Lämmel (CWI / Vrije Universiteit)
- Karina Olmos (Utrecht University)
- Eelco Visser (Utrecht University)
- Joost Visser (CWI)
- Hedzer Westra (Utrecht University)

Utrecht, April 2001

Contents

Chapter 1

Towards Typeful Stratego

Ralf Lämmel[1]

Abstract Stratego and the underlying system S are as yet untyped. We propose a type system which covers the essence of system S. In addition to system S, a generic traversal primitive for folding the children of a term is considered. This primitive is essential for type-changing traversal strategies. The type system which we propose is based on certain signature-independent generic types. We also have to introduce a few constructs which enable us to rephrase untyped Stratego programs in a typeful manner.

1.1 Preamble

We know how to type simple rewrite rules in Stratego. Let us think of a standard first-order many-sorted type system. We can also learn from other rewriting frameworks how to provide types for (some) rewriting strategies in Stratego. The type system of ELAN, for example, is pretty close to what we need for Stratego if we want to cover strategy combinators like $\cdot + \cdot$, $\cdot \leftarrow\!\!\!\!+ \cdot$, and $\cdot ; \cdot$. The hard part of typing Stratego or the underlying system S is to cover the generic traversal primitives like $\Box(\cdot)$. Therefore, we will focus on generic strategies, especially on concepts required for a corresponding type system. The extended abstract is largely driven by examples.

1.2 Examples

In Figure 1.1, we illustrate four examples (I)–(IV) of intentionally generic traversals. In (I), all naturals in the given tree are incremented as modelled by the rewrite rule $N \to succ(N)$. We need to turn this rule into a traversal strategy because the rule on its own is not confluent and terminating when considered as rewrite system. The strategy should be generic, that is, it should be applicable

[1]CWI, Kruislaan 413, NL-1098 SJ Amsterdam, Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam. `ralf@cwi.nl`, `http://www.cwi.nl/~ralf/`. The work of the author was supported, in part, by *NWO*, in the project "*Generation of Program Transformation Systems*".

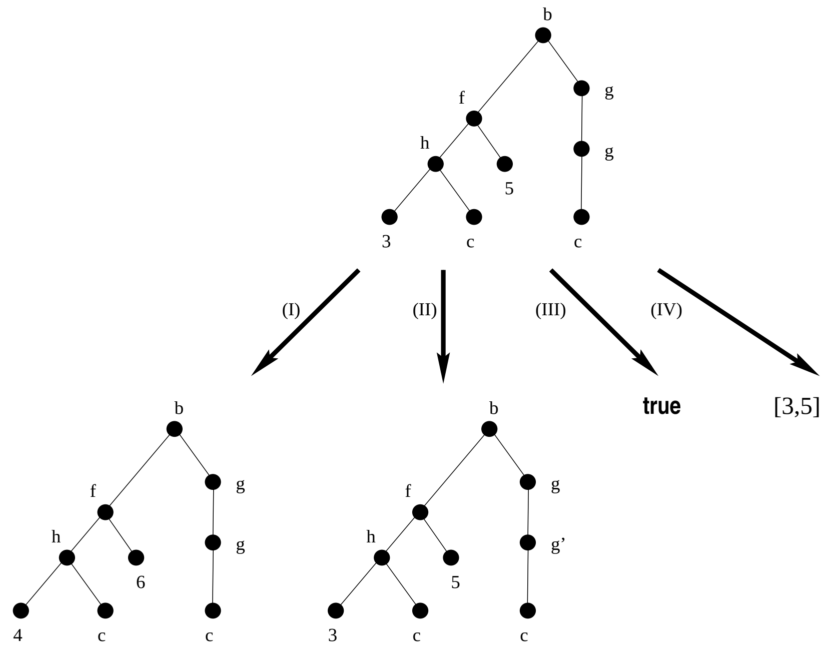

Figure 1.1: Generic traversals

to any term. In (II), a particular pattern is rewritten according to the rewrite rule $g(P) \to g'(P)$. Assume that we want to control this replacement so that it is performed in bottom-up manner, and the first matching term is rewritten. The strategy to locate the pattern is completely generic. (I) and (II) are examples of intentionally type-preserving strategies. We can easily solve these problems in untyped system S and Stratego by some strategy definitions. Note that we use *recursive* definitions rather than a recursive closure operator.

$$\begin{aligned} \mathit{natural} &= \mathit{zero} + \mathit{succ}(\textsc{Id}) \\ \mathit{traverse}_{(I)} &= (\mathit{natural}; N \to \mathit{succ}(N)) \nleftarrow \Box(\mathit{traverse}_{(I)}) \\ \mathit{traverse}_{(II)} &= \mathit{oncebu}(g(P) \to g'(P)) \end{aligned}$$

natural is an auxiliary strategy testing for naturals based on congruence strategies for *zero* and *succ*. Recall, $\mathit{traverse}_{(I)}$ is meant to increment all naturals. We use *natural* as a kind of dynamic type check to enable the applicability of the rewrite rule $N \to \mathit{succ}(N)$. $\mathit{traverse}_{(II)}$ finds the first pattern of form $g(x)$ in bottom-manner, and replaces it by $g'(x)$. Note the genericity of these traversals. They can be applied to any term. Of course, the strategies are somewhat specific because they deal with some constant or function symbols, namely *zero*, *succ*, g, and g'. Types for $\mathit{traverse}_{(I)}$ and $\mathit{traverse}_{(II)}$ will be provided in Section 3.

The examples (III) and (IV) in Figure 1.1 are examples of type-changing traversals, actually these are type-unifying traversals. In (III), we might test some property of the tree, e.g., if naturals occur at all. In (IV), we collect all the naturals in the tree using a left-to-right traversal. These strategies cannot faithfully be described in system S. By contrast, the strategies can be described in Stratego. The construct $\cdot\#\cdot$ offered by Stratego is essential to reduce children in a term in a generic manner. We are not going to rely on that operator because it is an obstacle to typing. We will later indicate a different but typeful operator to reduce children (Section 4 and Section 5). Therefore, we also postpone giving encodings of problems (III) and (IV).

1.3 Generic type-preserving strategies

Rewrite rules have *many-sorted* types. By contrast, generic strategies have *generic* types. Let us start with generic type-preserving strategies. The corresponding generic type is denoted by TP. Using a subtype relationship we can relate specific and generic types as follows: $\sigma \to \sigma \leq \mathsf{TP}$. Thus, the type TP subsumes all specific strategy types where input and output type are the same. It also means that a generic strategy can be applied to a specifically typed term. Let us consider the types of some strategy primitives. The constant strategy $\textsc{Id}$ has the type $\textsc{Id} : \mathsf{TP}$, i.e., it is a generic type-preserving strategy. The primitive $\Box(\cdot)$ has the type $\Box(\cdot) : \mathsf{TP} \to \mathsf{TP}$, i.e., in $\Box(s)$, the argument strategy s is a generic type-preserving strategy, and $\Box(s)$ itself is also a generic type-preserving strategy (which applies s to all children). The question is how can we qualify specific ingredients such as rewrite rules to become proper generic strategies. Given a rewrite rule r, we cannot just say $\Box(r)$ to apply r to all children because r has a specific type. Indeed, r does not know how to cope with terms of a different type than r's type. We might attempt to turn r into a generic strategy through the combinations $r \mathrel{\leftarrow\!\!\!+} \textsc{Id}$ or $r \mathrel{\leftarrow\!\!\!+} \textsc{Fail}$. This is not a typeful approach because $\cdot \mathrel{\leftarrow\!\!\!+} \cdot$ is designed for *choice based on success and failure*. Thus, the type of both ingredients for $\cdot \mathrel{\leftarrow\!\!\!+} \cdot$ should be the same. Also, specifically typed strategies would become generic to silently if the typing for $s_1 \mathrel{\leftarrow\!\!\!+} s_2$ would be the least upper bound of the types of s_1 and s_2. We need a new operator, say $\cdot \mathrel{\leftarrow\!\!\!\oplus} \cdot$, for *left-biased, type-sensitive choice*. In $s_1 \mathrel{\leftarrow\!\!\!\oplus} s_2$, the type of s_1 is more specific. So s_1 will be tried if the given term is of s_1's type. Otherwise, we resort to the generic default s_2. But note that s_2 will never be tried if s_1 is applicable no matter if s_1 succeeds or fails. Thus, $\cdot \mathrel{\leftarrow\!\!\!+} \cdot$ and $\cdot \mathrel{\leftarrow\!\!\!\oplus} \cdot$ embody different kinds of choice.

We redefine $\mathit{traverse}_{(I)}$ to obtain a typeful version:

$$\begin{aligned} \mathit{traverse}_{(I)} &= (\mathit{natural}; N \to \mathit{succ}(N)) \mathrel{\leftarrow\!\!\!\oplus} \Box(\mathit{traverse}_{(I)}) \\ &= N \to \mathit{succ}(N) \mathrel{\leftarrow\!\!\!\oplus} \Box(\mathit{traverse}_{(I)}) \end{aligned}$$

The simplification to eliminate the test for naturals is enabled by the typed model since the many-sorted rewrite rule sufficiently disambiguates the type of the specific ingredient. This solution illustrates that we can qualify specific rewrite rules to become generic by $\cdot \mathrel{\leftarrow\!\!\!\oplus} \cdot$. The other intentionally type-preserving strategy $\mathit{traverse}_{(II)}$ can be made fit in the same manner.

1.4 Folding children

Let us reconsider the problems (III) and (IV). We have to perform a kind of reduction. Stratego offers the $\cdot\#\cdot$ construct to access the children of a term as needed for reduction. The construct is similar to the univ operator (`=..`) of Prolog. These constructs have in common that children are essentially handled as lists. Since these lists are heterogeneous, there is little hope that we can type a construct like $\cdot\#\cdot$. We propose a different approach for accessing and processing children of a term. The main idea is that we never *explicitly* deal with children as lists, but we rather use a primitive to process the children. We suggest the primitive $(\!| s_\epsilon, s_\circ |\!)$ to perform a list-like right-associative fold on the children of a term. The first strategy parameter s_ϵ encodes the initial value for folding. In the case of a constant symbol, this strategy defines the result of folding. The second strategy parameter $s_\circ$ is used to compose a child with an intermediate value of folding.

In Section 5, we will explain how to type $(\!| \cdot, \cdot |\!)$ and derived strategies. A fundamental traversal strategy based on folding is the following:

$$crush(\nu, \nu_\epsilon, \nu_\pi) \quad = \quad \nu \nleftarrow (\!| \upsilon_\epsilon, (crush(\nu, \nu_\epsilon, \nu_\pi), \textsc{Id}); \nu_\pi |\!)$$

Note that we use the congruence strategy (s_1, s_2) for pairs in the strategy definition. $crush(s, s_\epsilon, s_\pi)$ performs a form of top-down reduction as follows. s is applied on the given term t. If s succeeds at some node, some data extracted from the node is supposed to be returned by s. If s does not succeed, a fold is performed to reduce the children. The two other parameter strategies can be conceived as the encoding of a monoid to be used for reduction. The data derived from crushing subtrees is composed with the strategy s_π which is meant to model the binary operation of the monoid for reduction. The strategy s_ϵ models the initial value for folding in the sense of the monoid's zero.

We can provide solutions to the problems (III) and (IV):

$$\begin{aligned} traverse_{(III)} &\quad = \quad crush(natural; N \to true, () \to false, or) \\ traverse_{(IV)} &\quad = \quad crush(natural; N \to cons(N, nil), () \to nil, app) \end{aligned}$$

$traverse_{(III)}$ reduces truth values by *or*. The neutral element is *false*. Encountering a natural, *true* is returned. $traverse_{(IV)}$ reduces lists by *app*ending them. The neutral element is the empty list *nil*. Encountering a natural N, a singleton list $cons(N, nil)$ is constructed from it. The straightforward definitions of *or* and *app* are omitted. Note the genericity of these traversals. The only assumption of these strategies is that they "know" of the signature for naturals.

1.5 Generic type-unifying strategies

The consideration of type-changing (and especially type-unifying) strategies leads us to other generic types than TP. The generic type $\mathsf{TU}(\sigma)$ denotes all type-unifying strategies, i.e., strategies which map all types to a fixed type σ. Subtyping is updated accordingly: $\sigma' \to \sigma \leq \mathsf{TU}(\sigma)$. To provide a type for $(\!| \cdot, \cdot |\!)$, we need another auxiliary type $\mathsf{TF}(\sigma)$ for its second parameter. Recall that the second parameter works on pairs in the sense that it takes a child (of any type) and an intermediate result of folding (of type σ) and composes them (to a value of type σ). Subtyping is updated accordingly: $(\sigma', \sigma) \to \sigma \leq \mathsf{TF}(\sigma)$.

The types of $(\!| \cdot, \cdot |\!)$ and the derived strategy *crush* are as follows:

$$\begin{aligned} (\!| \cdot, \cdot |\!) &\; : \; (() \to \sigma) \times \mathsf{TF}(\sigma) \times \mathsf{TU}(\sigma) \\ crush &\; : \; \forall \alpha.\ \mathsf{TU}(\alpha) \times (() \to \alpha) \times ((\alpha, \alpha) \to \alpha) \to \ \mathsf{TU}(\alpha) \end{aligned}$$

The type of *crush* illustrates that we also need to cope with polymorphism. The strategy *crush* is polymorphic in the sense that the result type of crushing is variable. The monoid parameters of *crush* determine this type. By contrast, the strategy *crush* is generic in the sense that the type of the term to be processed is arbitrary. We should not confuse polymorphism and genericity. A basic assumption for a polymorphic abstraction is that it behaves the same for all specific types. This is usually not the case for generic strategies because of the specific ingredients.

We re-encounter the problem how to turn a strategy s of a specific type into a generic one. So what is the generic default for a type-changing strategy s? ID and FAIL were applicable for type-preserving strategies, but they are not applicable for type-changing strategies because ID and FAIL themselves are type-preserving. However, we can think of a generic concept of failure, say FAIL_σ subscripted with the result type σ. Of course, this is not sensible for identity. Given a strategy s of a specific type $\sigma' \to \sigma$, we can turn this strategy into a generic type-unifying strategy $\mathsf{TU}(\sigma)$ by a new lifting operation $s \Uparrow \pi$ where π is a generic strategy type. We can think of $s \Uparrow \mathsf{TU}(\sigma)$ as $s \mathrel{\leftarrow\!\!\!\oplus} \text{FAIL}_\sigma$.

Let us rephrase the solution of problem (III) in a typeful manner. The traversal was based on *crush*. The first parameter of *crush* is meant to identify the data to be reduced. Potential failure is perfect for this identification process because it means that a given term does not expose a relevant pattern. The strategy $\mathit{traverse}_{(III)}$ as originally proposed is not typeable because the first parameter is instantiated with a specific strategy. We need to insert applications of $\cdot \Uparrow \cdot$. Also, the dynamic test for naturals becomes obsolete in our typeful setting if we assume many-sorted variables which in turn sufficiently disambiguate the type of the strategy. Finally, we need to perform type application because of the type parameter of *crush*.

$$\mathit{traverse}_{(III)} \quad = \quad \mathit{crush}[\mathit{Bool}](N \to \mathit{true} \Uparrow \mathsf{TU}(\mathit{Bool}), () \to \mathit{false}, \mathit{or})$$

The other intentionally type-unifying strategy $\mathit{traverse}_{(IV)}$ can be made fit in the same manner.

1.6 Plan

What needs to be done to design and to develop a typeful Stratego?

- We start from first-order many-sorted strategies as in ELAN.
- Generic traversal primitives $\Box(\cdot)$, $\Diamond(\cdot)$, and $\Box\!\!\!\!\Diamond(\cdot)$ are typed using TP.
- We do not include $\cdot\#\cdot$. We rather include $(\!|\cdot,\cdot|\!)$.
- $(\!|\cdot,\cdot|\!)$ is typed using $\mathsf{TU}(\sigma)$ and $\mathsf{TF}(\sigma)$.
- Strategy application has to be type-sensitive for $\cdot \mathrel{\leftarrow\!\!\!\oplus} \cdot$ and $\cdot \Uparrow \cdot$.
- Polymorphism needs to be accomplished.
- Special support for tuples has to be added to cope with multi-parameter strategies.
- The impure constructs of Stratego not present in system S need to be typed.

- We will have to decide what mixture of type-checking and type-inference we want.
- We should understand the limitations of the present proposal.
- At some point we might want to migrate the Stratego library.

Chapter 2

Functional Stratego

Eelco Dolstra[1]

Abstract Stratego is a domain-specific language intended for the construction of program transformation systems. To that end, the language has a number of interesting features; notably, first class pattern matching and generic traversal mechanisms. The question arises whether and how such features can be implemented in functional programming languages and Haskell in particular. In a rewrite system it is essential that one can easily recover from pattern match failure, so that alternative rewrite rules can be tried. Typical functional languages do not support this very well since local pattern match failure leads to global divergence. This article shows how to solve this problem by adding a choice operator to a simple functional language. It seems that this approach is not only useful to strategic programming, but is also more powerful than previous proposals to extend Haskell's pattern matching, such as views, patterns guards, and transformational patterns. Furthermore, we discuss how the generic programming techniques can be employed to implement generic traversals.

2.1 Introduction

Program transformation systems transform a computer program, typically represented as an abstract syntax tree, from one language to (possibly) another language. Examples of such systems are code generators, optimizers, and application generators (translating a high-level specification into a program in a general purpose language).

There are a number of features that seem to be especially important to the implementation of program transformations systems:

- Pattern matching, so that we can easily deconstruct and inspect values. More importantly, we need to have *first class* pattern matching: if a pattern fails to match, it should be easy to try alternatives.

- Traversal mechanisms. Transformations often need to applied at many points in the abstract syntax tree. We should have generic operations that can traverse arbitrary data structures, applying arbitrary transformations.

[1]edolstra@students.cs.uu.nl

Stratego [7] is a domain-specific language intended for the construction of program transformation systems, and as such has extensive matching and traversal features.

The question arises whether and how such features can be implemented in functional programming languages. After all, functional languages like ML and Haskell [4] are well-appreciated for their pattern matching facilities, and recursion makes it relatively easy to traverse a structured value. Unfortunately, pattern matching is not first class: pattern match failure leads to divergence (i.e., a crash) of the entire program. Also, Stratego is untyped, which makes generic traversal primitives like `all` possible; but in typed functional languages, this becomes problematic.

This article proposes a solution to the first problem by means of a choice operator which evaluates to its left argument, unless it fails; otherwise it evaluates to its right argument.

Outline. Section 2.2 explores the problem space of implementing Stratego-like constructs in Haskell. The addition of a choice operator is discussed in section 2.3, which also compares this to other proposals to extend pattern matching. Finally, section 2.4 briefly discusses generic traversals. The reader should be familiar with Stratego and Haskell.

2.2 Strategic programming in Haskell

It is not difficult to implement Stratego-like functionality. It would be nice if we could write a Stratego rewrite rule like:

```
PlusZero:  Plus(x, Const(0)) → x
```

as

```
plusZero = λ(Plus x (Const 0)) → x                              (2.1)
```

Of course, this will not work because we cannot handle pattern match failure in λ-abstractions. So we wrap the result in a `Maybe` type:

```
plusZero (Plus x (Const 0)) = Just x
plusZero _ = Nothing
```

In general, then, a strategy has the following type:

```
type Strat α β = α → Maybe β
```

Stratego's left choice and composition operators (here written as `<*`) are trivial:

```
(<+) ::  Strat α β → Strat α β → Strat α β
s1 <+ s2 = λt → maybe (s2 t) Just (s1 t)

(<*) ::  Strat α β → Strat β γ → Strat α γ
s1 <* s2 = λt → maybe Nothing s2 (s1 t)
```

Combining strategies is just as easy as in Stratego. Writing the basic rewrite rules, however, is tiresome because we spend a lot of time packing and unpacking the `Maybe` wrapper. For example, the congruence over a constructor `Add Expr Expr` would look like this:

```
cgrAdd s1 s2 (Add e1 e2) = maybe Nothing (λe1' →
  maybe Nothing (λe2' → Just $ Add e1' e2') (s2 e2)) (s1 e1)
```

Another issue is how to implement generic traversals. We have to describe explicitly for each data type how to traverse over it. Type classes can be used to make this generic:

```
class Trav α where
  allS ::  Strat α α → Strat α α
```

Functions like `bottomup` can then be defined in the usual way:

```
bottomupS ::  Trav α ⇒ Strat α α → Strat α α
bottomupS s = allS (bottomupS s) <* s
```

An instance might look like this:

```
instance Trav Expr where
  allS s (Const n) = Just $ Const n
  allS s (Add e1 e2) =
    case s e1 of
      Just e1' →
        case s e2 of
          Just e2' → Just $ Add e1' e2'
          Nothing → Nothing
      Nothing → Nothing
```

There are more refined (and complex) approaches (see [5]), but these also suffer from the fact that a lot of code must be written for each data type over which we want to traverse.

2.3 Extending the λ-calculus with a choice operator

2.3.1 Semantics

In the previous section, example 2.1 did not work because it is not possible to deal with pattern match failure in the left-hand side of λ-abstractions. In this section I shall give the semantics of a simple (untyped) functional language extended with a choice operator which first tries to evaluate its left argument, and if that fails, evaluates its right argument.

The syntax is the λ-calculus extended with constructed values, pattern matching λ-abstraction, and a choice operator:

$$e ::= x \mid ee \mid C \mid \lambda p \rightarrow e \mid e \texttt{ <+ } e \mid \delta$$
$$p ::= x \mid pp \mid C$$

where x ranges over the variables, and C over the constructors. The constant δ denotes pattern match failure. The choice operator `<+` binds weaker than λ-abstraction and function application.

The semantics is given by the following rewrite rules:

$$\begin{array}{rcll}
(\lambda x \rightarrow e_1)e_2 & \Rightarrow & [x := e_2]e_1 & (\beta\text{-reduction}) \\
(\lambda C \rightarrow e_1)C & \Rightarrow & e_1 & (\text{pos. constr. match}) \\
(\lambda C_1 \rightarrow e_1)C_2 & \Rightarrow & \delta \text{ (if } (C_1 \neq C_2)) & (\text{neg. constr. match}) \\
(\lambda(p_1 p_2) \rightarrow e_1)(e_2 e_3) & \Rightarrow & ((\lambda p_1 \rightarrow \lambda p_2 . e_1)e_2)e_3 & (\text{application match}) \\
e_1 \texttt{ <+ } e_2 & \Rightarrow & e_1 \text{ (if NF}(e_1) \neq \delta) & (\text{left choice}) \\
\delta \texttt{ <+ } e & \Rightarrow & e & (\text{right choice}) \\
\delta e & \Rightarrow & \delta & (\text{failure propagation})
\end{array}$$

where NF(e) denotes the normal form of term e. These rules suffice to implement a lazy evaluator (we have done so in Stratego for a system extended with let-bindings).

The semantics is given as a set of rewrite rules from the language to the language, i.e., as source transformations. The advantage is that no additional syntax or notation is required. Furthermore, such rules can be employed directly in e.g. an optimizer.

Unfortunately, something seems to be missing. For example, we would like to define the `if` function as follows:

```
if = λTrue e1 e2 → e1 <+ λFalse e1 e2 → e2                (2.2)
```

expecting that `if False 1 2` $\Rightarrow$ `2`. This is, after all, the behaviour expected from Stratego. However, this obviously will not work, since in the evaluation of `if` the choice operator will immediately pick the *left* side, since `λTrue e1 e2 → e1` $\neq \delta$.

We can get around this by writing instead:

```
if = λc e1 e2 → (
    (λTrue e1 e2 → e1) c e1 e2 <+                           (2.3)
    (λFalse e1 e2 → e2) c e1 e2)
```

This will enforce that `c` is matched against the pattern `True`, and so the left side will fail and the right side will be evaluated instead.

A nicer solution (which is admittedly somewhat *ad hoc*) is to add the following rule:

$$(\lambda p \rightarrow e_1) \texttt{ <+ } e_2 \quad \Rightarrow \quad \lambda x \rightarrow ((\lambda p \rightarrow e_1)x \texttt{ <+ } e_2 x) \quad (\text{distr. arguments})$$

and to add the restriction to the left choice rule that e_1 must not be a λ-abstraction.

In other words, this rule pushes arguments into the choice operands until we have a base type (a constructed value). It is easy to see that this rule automatically transforms example 2.2 into example 2.3.

We don't lose any expressive power here. Even if we actually want to choose between functions and not results of functions (e.g. in `(if b then f else g) <+ h`), where `b` may fail, we can do this by wrapping the function in a constructor (e.g. `(if b then F f else F g) <+ (F h)`) and unpacking the result later, outside of the scope of the choice operator.

A problem with the choice operator is that we do not have control over the scope of the failure handling, e.g. in `(`λ`Foo` $\rightarrow$ `e) <+ ...` we might want the choice operator to catch failure in the `Foo`-match but not in `e`. This problem also confronts Stratego programmers. For example, if we want to rewrite `Foo`-terms and nothing else, we might write in Stratego `try(?Foo; s)`. Unfortunately, any rewrite failure in `s` (including those due to programming errors) will be caught indiscriminately by `try`.

2.3.2 Comparison to other approaches

The question is what the choice operator gains us, apart from nicer strategic programming. It is well known that regular pattern matching is not perfect [6, 2]. It turns out that the choice operator eliminates the need for many of the proposals to extends Haskell's pattern matching, such as *views*, *pattern guards*, and *transformational patterns*. Furthermore, it makes Haskell's "equational style" (writing a function definition as a number of pattern-guarded equations which must be tried one after another) unnecessary, as well as `case`-expressions.

Equational style unnecessary. Haskell allows us to write patterns not just in λ-abstractors but also in function definitions:

$$\begin{array}{l}
\texttt{f}\ p_{11}\ \ldots\ p_{1n}\ \texttt{=}\ e_1 \\
\texttt{f}\ p_{21}\ \ldots\ p_{2n}\ \texttt{=}\ e_2 \\
\ldots \\
\texttt{f}\ p_{m1}\ \ldots\ p_{mn}\ \texttt{=}\ e_m
\end{array}$$

The semantics here are different than in λ-abstractions: if a pattern match fails, the program does not diverge, but instead the next equation is tried. This is an *ad hoc* mechanism that becomes redundant if we have a choice operator:

$$\begin{array}{l}
\texttt{f =} \\
\quad \lambda p_{11}\ \ldots\ p_{1n} \rightarrow e_1\ \texttt{<+} \\
\quad \lambda p_{21}\ \ldots\ p_{2n} \rightarrow e_2\ \texttt{<+} \\
\quad \ldots \\
\quad \lambda p_{m1}\ \ldots\ p_{mn} \rightarrow e_m
\end{array}$$

In fact, a major advantage is that pattern matching λ-abstractors are now first class, so we can write:

$$\begin{array}{l}
\texttt{f}_1\ \texttt{=}\ \lambda p_{11}\ \ldots\ p_{1n} \rightarrow e_1 \\
\texttt{f}_2\ \texttt{=}\ \lambda p_{21}\ \ldots\ p_{2n} \rightarrow e_2 \\
\ldots \\
\texttt{f}_m\ \texttt{=}\ \lambda p_{m1}\ \ldots\ p_{mn} \rightarrow e_m \\
\texttt{f = f}_1\ \texttt{<+ f}_2\ \texttt{<+ ... <+ f}_m
\end{array}$$

and we can combine the f_i arbitrarily.

Case-expressions can be eliminated from the language in a similar way.

Views. Views [8] were proposed to address the problem that regular pattern matching is rather limited since we can only match with actual constructors. As a consequence we cannot match against e.g. the end of a list instead of the head, nor can we match against abstract data types since there is simply nothing to match against. For example, using the proposed views for Haskell [1] we can write the following view to match against the end of a list:

```
view Tsil a of [a] = Lin | Snoc y ys where
  tsil xs =
    case reverse xs of
      [] → Lin
      (y:ys) → Snoc y ys
```

where matching against a `Snoc`-constructor causes the function `tsil` to be applied to the value:

```
f (Snoc y _) = y
f Lin = 0
```

It is worth pointing out why views (and transformational patterns) are useful. The reason is that the equational style can only be used if the non-applicability of an equation can be discovered in the pattern. When that is not possible, the equational style falls apart, and we have to explicitly write the traversal through the alternatives (the equations) as a series of ever deeper nested `case`-expressions. The choice operator liberates us from this regime, hence the main motivation for views and transformational patterns disappears. With it, the previous example becomes:

```
f = (λ(y:_) → y <+ λ[] → 0) .  reverse
```

Views still have the advantage that the transformation to be applied (e.g. `tsil`) is implicit in the name of the patterns (e.g. `Snoc`), but this seems only a minor advantage.

Pattern guards. In Haskell's equational notation, we can use boolean guards to further restrict the applicability of an equation, e.g. `f x | x > 3 = 123`. However, there is a disparity between patterns and guards: patterns can bind variables, whereas guards cannot. For example, if we want to return a variable from an environment, or 0 if it is undefined, we would write:

```
f env var  |  isJust (lookup env var)
           =  fromJust (lookup env var)
f env var  =  0
```

where `lookup` has type `[(`α`, `β`)] → `α` → Maybe `β. This is awkward because we now inspect the result of `lookup` twice. Pattern guards [2] redefine a guard as a list of qualifiers, just like in a list comprehension, so that binding can occur:

```
f env var | Just x ← lookup env var = x
f env var = 0                                  (2.4)
```

But when we have a choice operator, we can just write:

```
f env var = let (Just x) = lookup env var in x <+ 0
```

Alternatively, we could just get rid of the `Maybe` result of `lookup` altogether, making it of type `[(`α`, `β`)] → `α` → `β, and then we get:

```
f env var = lookup env var <+ 0
```

Transformational patterns. Transformational patterns [2] provide a cheap alternative to views, allowing us to write example 2.4 as:

```
f env (Just x)!(lookup env) = x
f env var = 0
```

Hence, transformational patterns are just view transformations made explicit. The choice operator allows a similar notation:

```
f env = (λ(Just x) → x) .  (lookup env)
  <+ λ_ → 0
```

In general, a definition `f pat!fun = e` can be written as `f = (λpat → e) . fun`.

2.4 Generic traversals

Given the untyped calculus given in the previous section, it is not hard to define the `all` primitive (which applies a function to each subterm of a term):

```
all  = λf →
     ( λ(c x) → (all f c) (f x)
     <+ λc → c
     )
```

But it would be hard to type this function. Previous work on generic programming, such as Generic Haskell [3], seems not to help us here. For example, in Generic Haskell, we write a generic function by giving a clause for binary sums, binary products, and primitive types, since all data types can be decomposed into a combination of these. Unfortunately, this moves us below the level of constructors. For example, it is not hard to give the primitive clause for an implementation of `bottomup`:

```
bottomup(t) ::  (t → t) → t → t
bottomup(1) f x = f x
```

But we cannot give a clause for e.g. binary products: it is clear that we should apply `bottomup` to the arguments, but then what? At some point `f` should be applied, but only if the binary product is in fact of type `t`, and not just part of something of type `t`.

```
bottomup(a * b) f (x, y) =
  ???  $ (bottomup(a) f x, bottomup(b) f y)
```

It also seems impossible to implement `all`, since a generic function traverses the entire value, not just the top-level subterms.

2.5 Conclusion

We have discussed two features that seem to be useful to strategic programming in a functional language. First, there is the proposed mechanism to handle pattern match failure, which has the additional advantage of making some proposed extensions to pattern matching redundant. Second, generic traversals are important, but it is not currently clear how these can be implemented elegantly.

References

[1] Warren Burton, Erik Meijer, Patrick Sansom, Simon Thompson, and Phil Wadler. Views: An extension to haskell pattern matching. `http://www.haskell.org/development/views.html`.

[2] Martin Erwig and Simon Peyton Jones. Pattern guards and transformational patterns. In *Haskell Workshop*, 2000.

[3] Ralf Hinze. A generic programming extension for Haskell. In *Haskell Workshop*, 1999.

[4] Simon Peyton Jones, John Hughes, et al. Report on the programming language Haskell 98, 1999.

[5] Ralf Lämmel and Joost Visser. Type-safe functional strategies. In *Scottish Functional Programming Workshop*, July 2000.

[6] Mark Tullsen. First class patterns. In *2nd International Workshop on Practial Aspects of Declarative Languages*, number 1753 in LNCS, pages 1–15, 2000.

[7] Eelco Visser, Zine el Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *International Conference on Functional Programming (ICFP'98)*, ACM SIGPLAN, pages 13–26, September 1998.

[8] Phil Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *14th ACM Symp. on Principles of Programming Languages*, pages 307–313, 1987.

Chapter 3

XT Capita Selecta

Merijn de Jonge and Joost Visser

Abstract XT is a bundle of program-transformation tools. Stratego is part of this bundle, and is used as implementation language for many tools throughout its packages.

Giving special attention to the role of Stratego, we discuss a selection of our XT experiences. These range from the construction of meta-tools to support Stratego programming, through Stratego techniques applied in constructing some of XT's constituents, to projects where XT (including Stratego and tools programmed in Stratego) has been applied to program-transformation problems.

3.1 Stratego and xt

Overview of xt XT is a bundle of program-transformation tools [3]. Its purpose is to support component-based development of program transformations. Figure 3.1 gives an overview of the individual tool packages that are bundled by XT. At the heart of XT lies the ATERM library [1]. The ATERM format is a generic tree representation format, for which the ATERM library provides space and time efficient support. Within XT, the ATERM format is used as a common exchange format for parse trees and abstract syntax trees.

The Stratego package contains the Stratego compiler [12] and the Stratego standard libraries [13]. ATERMS are used both as input and output of Stratego programs and internally for term representation.

The syntax definition formalism SDF [5, 11] is used throughout XT as grammar formalism. The packages `pgen` and `sglr` contain the parse table generator and generic parser that support SDF. The pretty-print table generator and generic pretty-printer for SDF are provided by the GPP package [2]. All these tools make use of ATERMS as exchange and representation format.

The JJForester package contains a code generator to support representation and traversal of tree structures in Java [9]. Again, the ATERMS are used to represent and exchange trees.

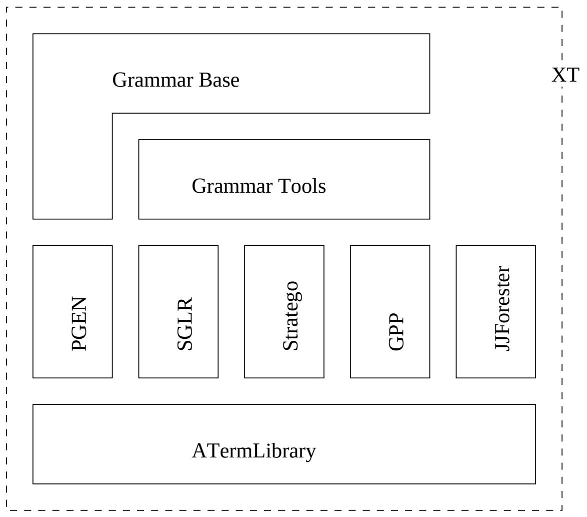

Figure 3.1: XT is a bundle of packages.

The GrammarTools package contains a suite of tools for grammar analysis, grammar (re)construction, and (parse) tree manipulation. Most of these tools are built from components programmed in Stratego, and instantiations of `sglr` and the generic pretty-printer. Again, ATerms are used for tree exchange between these components.

The Grammar Base is a collection of syntax definitions in SDF for a growing number of formats, specification languages, and programming languages. The Grammar Base uses `pgen`, `sglr`, and GPP, and various constituents of the GrammarTools.

The philosophy of xt In the design and organization of XT three guiding principles have been followed to facilitate component-based transformation development.

Many, small, stand-alone components The functionality offered by the various packages that XT bundles should as much as possible be available in separate chunks that can be individually (re-)used. Integration of individual pieces of functionality can be left to the user of the package, or can be offered in such a way that separate use of the components is not obstructed.

Simple, common exchange format The input and output of all components should as much as possible be done in a simple, common exchange formats. In XT, ATerms are used for this purpose. Having a common exchange format ensures smooth interoperation of components from different packages.

Grammars as contracts The shape of (parse) trees that are represented in the exchange format should be defined in grammars that are independent of specific components, and independent of specific implementation languages. Code for tree representation, for reading and writing trees, for

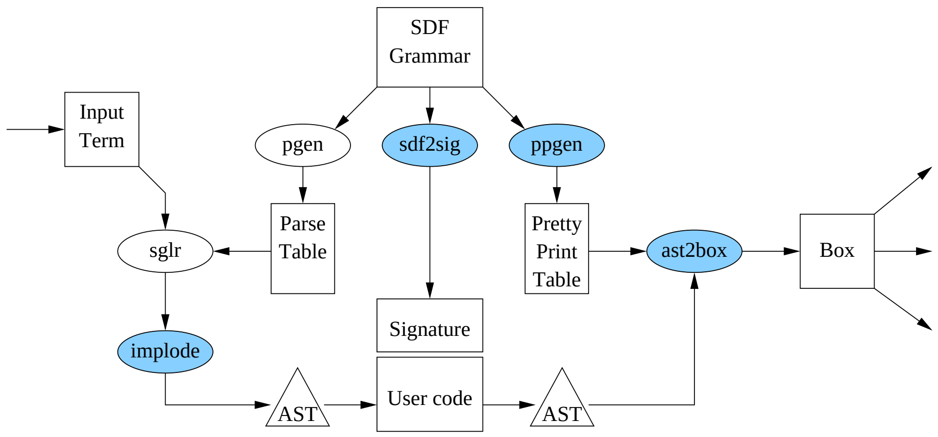

Figure 3.2: Meta-tools for parsing, signature generation, and pretty-printing integrate Stratego with SDF.

> parsing and for pretty-printing should be generated from grammars as much as possible. Thus, grammars should be used as component interface definitions (contracts).

The role of Stratego in xt As the overview of XT already suggests, Stratego fulfills several roles in XT. Firstly, Stratego is a constituent of XT that can be used by transformation developers as a powerful transformation language. Secondly, it is used within XT itself as implementation language for tools in various packages. For instance, almost all grammar tools are implemented in Stratego, as are several components of GPP. Finally, Stratego programming is supported by a number of meta-tools contained in the GrammarTools package.

In the upcoming sections we will shed light on these roles. In Section 3.2 we explain the Stratego meta-tooling contained in XT. In Section 3.3 we discuss a selection of the constituents of XT programmed in Stratego. In Section 3.4 we touch on some programming and engineering techniques related to Stratego that we deployed in the development of XT. Finally, Section 3.5 evaluates Stratego, in light of its use in XT, and lists some suggestions for further development of the language and its support.

3.2 Meta-tooling

Apart from the Stratego compiler, a number of meta-tools are available in XT that support Stratego programming. These include tools that integrate Stratego with SDF, and tools that analyze the import structure of Stratego programs.

Integration with sdf Figure 3.2 shows an overview of the meta-tools offered by XT that integrate Stratego and SDF. In combination, these tools instantiate a meta-tooling architecture where grammars are used as contracts between components [4]. The shaded ellipses are tools that are themselves programmed in Stratego.

Normally, the Stratego programmer manually constructs the signatures he needs. Also, the input and output of his programs are ATERMs. With the meta-tooling offered by XT, the programmer can construct front and back-ends to his programs for parsing and pretty-printing, and he can generate the signatures he needs from the grammars he uses. In general, the programmer follows the following steps:

- The programmer constructs or reuses an SDF grammar. From this grammar, he generates a parse table, a signature, and a pretty-print table, using the tools `pgen`, `sdf2sig`, and `ppgen`.
- The programmer imports the generated signature into his program.
- A concrete input term is fed into the generic parser `sglr`, together with the generated parse table. The resulting parse tree is imploded to an abstract syntax tree (AST) using the `implode-asfix` tool. As this AST is represented by an aterm, it can be consumed by the Stratego program.
- The output of the Stratego program is pretty-printed in two steps. First, the AST is transformed into a formatted BOX expression by `ast2box`, using the formatting rules in the generated pretty-print tables (possibly supplemented with customizations of the user). Secondly, the box expression is transformed to plain text, HTML, or LaTeX. This is done with the BOX back ends, which are not pictured.

Note that the parse and pretty-print tables are not Stratego-specific. Consequently, the parsing and pretty-print back-ends can be used for any component that consumes and produces aterms. Analogous to `sdf2sig` for Stratego, SDF-driven code generators exist for other programming languages, e.g. JJForester for Java. Stratego programs can seamlessly interoperate with programs in those languages, because they use code generated from the same grammar and employ the ATERM format as exchange format.

Import analysis Using Stratego's import mechanism, Stratego programs can be built from several modules. For various purposes, the import relations that exist among modules need to be analyzed. One of these purposes is compilation. In fact, one of the earliest phases of the Stratego compiler is *packing*, i.e. collecting all imported modules into a single file. The `pack-stratego` component, which takes care of packing, is actually a Stratego-specific instantiation of the generic meta-tool `pack`, which is part of XT. Another instantiation is `pack-sdf`, which collects SDF modules into a single SDF definition file. Apart from a file with packed modules, the `pack` tool also produces a dependency file, which can be included in a Makefile.

Another purpose for which import analysis is useful is program analysis and program comprehension. For these purposes, XT offers another generic tool, called `import`. When instantiated for Stratego, it analyses the import relations for a given program, and presents the analysis results as a list of module names, a list of full file names, or an import dependency graph. For instance, the generated import graph for the Stratego library `lib` is show in Figure 3.3. Note that this picture immediately reveales two mutual imports (of `lib` and `parse-options`, and of `list` and `list-filter`) that are probably unintended.

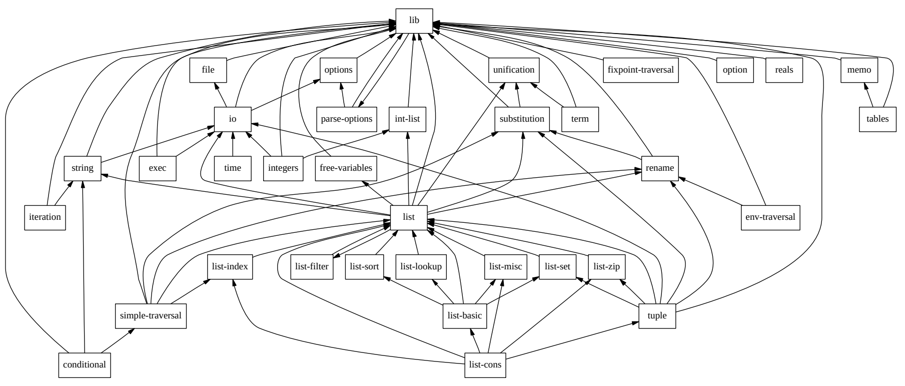

Figure 3.3: Generated import graph for the Stratego library.

Both `pack` and `import` make use of the generic graph strategies that are contained in the Stratego library. These two tools have much functionality in common. We consider combining them in a single tool that is not only generic with respect to the input language, but also with respect to the actions performed after analysis: result presentation or packing (several dimensions of genericity).

3.3 xt constituents

Apart from the above-mentioned meta-tools for Stratego, XT contains a wide variety of transformation components that have been programmed in Stratego. We will explain a selection of these components for graph transformation, tree transformation, and grammar transformation.

3.3.1 Graph transformation

Graph formats Graph formats and corresponding tooling play an important role within XT for code analysis and code visualization. According to the component based idea, XT reuses existing graph visualization tooling based on the `dot` input format [8]. This format allows graphs to be defined easily by simply specifying the nodes and edges of the graph, and it provides the ability to influence the visual appearance by specifying additional properties of the graph and its nodes and edges. XT also uses the GraphXML [6] graph format. This XML format provides the ability to transfer graphs between XT and third-party tooling and allows for instance to re-use existing XML tooling for the implementation of graph transformations. XT also offers the necessary conversion tooling to transform a graph represented in GraphXML to `dot`. For both formats SDF grammars are available in the Grammar Base and with XT's meta-tools mentioned above, parsers, pretty-printers and Stratego signatures have been generated. These have been used to construct graph transformation components.

Converters To convert between graph representation formats, several graph transformations have been programmed, including `GraphXML2dot`.

Analysers The Stratego module `GraphXML-analysis` contains a number of reusable strategies that perform analysis on graphs represented in GraphXML. The transformation component by the same name performs a number of these analysis strategies, and presents the results as plain text. Most of these analyzers are implemented using the standard set strategies as contained in the Stratego library in combination with `collect`. For example, the strategy to obtain the source nodes of a graph, was implemented as follows:

```
get-sources = collect( \ source(x) -> <de-quote>x \ )
```

Similarly, target nodes can be obtained using the `get-targets` strategy, implemented as:

```
get-targets = collect( \ target(x) -> <de-quote>x \ )
```

Both strategies can be combined to form more advanced analyzers. To obtain those nodes from a graph with only in-going edges (sinks), the `get-sinks` strategy was implemented:

```
get-sinks
  = \g -> <diff>(targets,sources)
     where <get-targets>g => targets
         ; <get-sources>g => sources
    \
```

Extractors A number of components are available in XT for extracting graphs from source code. An example is `sdf2sg`, which extracts a sort dependency graph (or sort graph) from an SDF grammar. The extracted graph is represented in GraphXML. The utility `get-sort-graph` is a script that glues the extractor `sdf2sg` with the converter `GraphXML2dot` and the graph formatter `dot`.

3.3.2 Tree transformation

Tree visualization The component `treeviz` takes an arbitrary ATerm, and produces a visualization of it in the GraphXML format. This is accomplished by decomposing each term $f(a_1, \ldots a_n)$ using the '#' construct of Stratego and creating edges $f \rightarrow a_i$ for each argument ter $a_1 \ldots a_n$:

```
NodeToEdges =
   ?f#( args );
   !args;
   map(NodeToEdge(!target(<quote>f)))

NodeToEdge(target) =
   ?f#(_);
   !edge([source(<quote>f), <target>()] )
```

Parse tree implosion The component `implode-asfix` takes a parse tree in the AsFix format, and produces a slimmed down syntax tree. Various command line options allow control over which implosion filters are activated. When all filters are active, the parse tree is transformed into an abstract syntax tree, which contains no layout or literals, and in which lexical parse trees have been flattened to strings.

The `implode-asfix` component demonstrates the use of command line options in Stratego and the sequential application of filters to an input term. To define command line options in addition to the standard set of options, we use the strategy `iowrap0` and pass it all additional options. The standard Stratego options do not need to be passed to `iowrap0`.

```
main =
   iowrap0( implode, Option( "--lex",    !FlatLex )
                   + Option( "--layout", !RemoveLayout )
                   + Option( "--lit",    !RemoveLit )
                   + Option( "--appl",   !ReplaceAppl )
                   + Option( "--inj",    !FlatInj )
                   + Option( "--list",   !FlatList )
                   + Option( "--seq",    !RemoveSeq )
                   + Option( "--pt",     !RemovePT ) )
```

The `iowrap0` strategy passes a tuple to its argument strategy (`implode` in the example) consisting of the list of command line options as specified by the user and the input term. The strategy should also return a tuple with the arguments and the transformed term. The strategy `implode` accesses the command-line options to determine which filter to apply to the input term. The strategy `option-defined` is used to check whether an option was specified or not. On success, a particular filter is applied, on failure the next option is checked.

```
implode =
   ?term;
   try((option-defined(FlatLex),      flat-lex));
   try((option-defined(RemoveLayout), remove-layout));
   try((option-defined(RemoveLit),    remove-lit));
   try((option-defined(FlatList),     flat-list));
   try((option-defined(ReplaceAppl),  replace-appl));
   try((option-defined(FlatInj),      flat-injections));
   try((option-defined(RemoveSeq),    remove-seq));
   try((option-defined(RemovePT),     remove-pt));
   try(?term; (id, implode-asfix))
```

When no options were specified, the input term remains unchanged after all option checks have been performed. The last match in the example, which tries to match the same input term as the first match therefore succeeds, and a default implosion will be applied.

Syntax tree formatting To format parse trees and abstract syntax trees, Box front-ends `asfix2box` and `ast2box` are available. Apart from a tree, these utilities take a sequence of pretty-print tables as input. The output is a term in Box, a language independent markup language to describe the intended layout of text. A Box term can be passed to one of the Box back-ends to obtain HTML, LaTeX or plain text as output.

The Box front-ends use Stratego's table mechanism for efficient storage and retrieval of pretty-print rules. During initialization, a global accessible table is created with `create-table` and filled with all pretty-print rules defined in the pretty-print tables:

```
read-pp-tables =
   ?names;
   <create-table>"pp-table";
   <map(ReadFromFile; build-pp-table)>names
```

Pretty-print rules are inserted in the table with the `table-put` strategy:

```
StoreEntry =
   ?PP-Entry( path, template );
   <table-put>("pp-table", <mk-key>path, (path, template))
```

After initialization, the pretty-print rules are accessed with `table-get` to transform an abstract syntax tree or parse-tree to BOX:

```
pp-table-get =
   ?key;
   <table-get>("pp-table", key ) => (path, template)
```

3.3.3 Grammar transformation

Grammars play an essential role in XT. They are used to generate parsers, pretty-printers, Stratego signatures, and Java libraries for language-specific tree transformation. In fact, grammars are used to fixate the interfaces between transformation components [4]. To support this employment of grammars, we need tools to create, manipulate, and transform grammars, and to generate code from grammars. We discuss a few of these tools.

Syntax definition In XT, we use SDF as primary syntax definition formalism. Because of its purity and declarativeness, grammars written in this formalism are well suited to be used for different purposes (parsing, pretty-printing, code generation), as is desired in XT. Also, its modularity and expressiveness enables syntax reuse.

Grammar extraction Grammars can of course be written manually from scratch, but when a language definition in a different formalism is already available, an SDF definition can be generated in stead. The GrammarTools include several components for this purpose. The tool `yacc2sdf` extracts an SDF grammar from a Yacc parser description. The tool `happy2sdf` does the same for the Yacc-derivative Happy that targets Haskell. This latter tool is actually a composition of `happy2yacc` and `yacc2sdf`.

In XML, document type definitions (DTDs) are used to describe the structure of XML documents, i.e. to describe the abstract syntax to which they must conform. The component `dtd2sdf` extracts an SDF definition from such a DTD. From the extracted grammar, a validating[1] parser can be generated to parse XML documents that conform to the given DTD. From the same grammar, a DTD-specific Stratego signature can be generated, to support development of DTD-aware Stratego components.

Grammar rephrasing Rephrasings are transformations where the source and target language coincide, or largely overlap. Example of a grammar rephrasings are the tools `sdf2asdf` and `sdf-normalize`, which both map SDF to a subset of SDF.

Other examples of rephrasings are `sdf-cons` and `sdf-label`, which synthesise labels and constructor names from SDF productions, and add these to

[1]An XML parser that enforces a document's conformance to a DTD is called a *validating* parser.

a grammar, and the converse tools `rm-cons` and `rm-labels`. The latter tool makes use of Stratego's overlay construct to simplify programming on parse trees in the verbose parse tree format AsFix. The grammar of the SDF language contains the following production for labelled symbols:

```
  Literal ":" Symbol -> Symbol {cons("label")}
```

Using the meta-tool `sdf2overlay`, which is an enhanced version of `sdf2sig`, the following Stratego code can be generated:

```
signature
  constructors
    label : Literal * Symbol -> Symbol
overlays
  label-overlay(u_20,v_20,w_20,x_20)
    = appl(prod([cf(sort("Literal")),cf(opt(layout)),
      lit(":"),cf(opt(layout)),cf(sort("Symbol"))],cf(
      sort("Symbol")),attrs([cons("label"),'id(
      "Label-Sdf-Syntax")])), [u_20,v_20,appl(prod([
      char-class([58])],lit(":"),no-attrs),[58]),w_20,
      x_20])
```

The right-hand side of the generated overlay is the AsFix fragment that represents the concrete parse tree that represents labeled symbols. Given this generated overlay, the tool that removes labels is now simply programmed as follows:

```
  rm-labels
    = topdown( try(\label-overlay(_,_,_,d) -> d\ ) )
```

Thus, the overlay is used to match labeled symbols, select their subterm that represents the symbol without label, and replace them with this subterm.

3.4 Programming and engineering techniques

During the development of XT, we employed a range of programming and engineering techniques that are more or less specific to Stratego. Though these techniques are hardly novel, and certainly not earth-shaking, they might be of interest to other (potential) Stratego practitioners. For that reason, we discuss them.

Learning the ropes To learn Stratego, a wide range of sources is available. In our experience, the best starting place is the Stratego Tutorial [14], in combination with the Stratego Library [13]. To go beyond first principles, it is best to consult examples of systems programmed in Stratego, such as the Stratego Compiler itself [12]. We can also recommend the GrammarTools package of XT, as it contains a large number of fairly small components. The Stratego homepage is implemented as a Wiki server. There you can find (and add) information that is complementary to the official sources, and discussions about Stratego practice. If you really get stuck, you can post an SOS message on the Stratego mailing list.

Understanding and reusing code Reuse is better than programming from scratch. However, to reuse strategies, one first needs to understand them. In our experience, this can be quite difficult. To find out, for instance, how to invoke a strategy, one needs to know what kind of terms it expects, and what kinds of strategies are needed to instantiate its parameters. Lacking a type system or a documentation standard, such information is difficult to glean from the code.

To reconstruct such 'usage information', two complementary tactics can be followed: look at the definition of the strategy, or look at its call sites. In our experience, the second tactic is easier and more often successful. To find call sites of strategies, one can perform searches in code bases, or PDF documents.

Despite the lack of a proper mechanism for code understanding in Stratego, developing reusable code is simplified thanks to the modularization mechanism, argument strategies, and untypedness. Stratego's standard library provides a great number of reusable strategies which significantly decreases the amount of code that needs to be written for each application. Figure 3.3 depicts the import graph of the Stratego library and gives an impression of the diversity of he functionality of this library.

Choosing the proper construct Often, different language constructs can be chosen to express the same. Which construct to use depends on personal taste, there usually is no general rule to follow. Below we discuss a few of such constructs.

- A Stratego rule is syntactic sugar for a strategy which starts with a match and ends with a build construct:

  `L:l->r where s` is *equivalent* to `L={x1,..,xn: ?l;where(s);!r}`

  A Stratego rule looks similar to a rewrite rule, which transforms a term from left to right. This similarity can be used as heuristic to choose between a rule and a strategy: use a rule when a transformation is similar to a rewrite step.

- Arguments to rules and strategies can be passed through strategy variables or by tupling. For instance, the `iowrap0` discussed before uses two strategy variables, whereas the strategy `implode` that is called by `iowrap0` is passed a tuple with the specified command line options and the input term. Tupling arguments usually requires explicit matching and construction of the tuple as in the example below:

  ```
  ?aterm;
  <strategy_a>(arg1, ..., argi, aterm);
  ?result;
  <strategy_b>(argj, ..., argn, result)
  ```

  With strategy variables, the two matches are not required, and the code looks more like the sequential application of transformations:

  ```
  strategy_a(arg1, ..., argi);
  strategy_b(argj, ..., argn)
  ```

- To construct and de-construct terms in Stratego the match ('?') and build ('!') operators can be used. For example, to construct a term t' using a sub-term s occurring in a term t, the following Stratego construct can be used:

  ```
  ?t(..., s, ... );
  !t'( ..., s, ... );
  ```

  This can also be done using the anonymous rule construct:

  ```
  \ t(..., s, ...) -> t'( ..., s, ...) \
  ```

  This similarity allows rule-like definitions as strategies. See the definition of `get-sinks` for an example. It uses an anonymous rule in a strategy definition for matching the input term and building the output term.

- When the same term should be used at several places in a strategy definition, the term needs to be preserved before its first use until its last use. This can be achieved by assigning the term to a Stratego variable:

  ```
  ?new_term
  s_1;...;s_n
  !new_term
  ```

  In this example, the input term is saved before executing the strategies $s_1 \ldots s_n$ and restored afterwards. The same can be achieved without introducing additional variables using the `where` construct. For example, to search for a pattern in a term and assigning it to a variable `pattern` without affecting the input term, the following Stratego code can be used:

  ```
  where( oncetd(?f(a,b)=>pattern))
  ```

Debugging heuristics Debug support in Stratego is rather minimal, and consequently errors are usually hard to locate. A few simple techniques are available to help debugging Stratego programs, but debugging remains a time consuming business.

- Stratego offers the `debug` strategy, which writes the subject term to standard error. This strategy accepts a string as optional argument which is written before the subject term and helps to distinguish between different `debug` invocations. An often occurring mistake is to forget to build the argument string (by omitting the Stratego build operator). In this case no debug output is displayed and you might conclude incorrectly that the debug strategy is not reached because of a bug somewhere before. It is because of the incorrect use of `debug` however that no debug output is displayed, not necessarily due to a bug in your program.

- Often you believe that a subject term matches a particular pattern but somehow a strategy operating on that term fails. Is the strategy incorrect, or does the input term not match the expected pattern? An input term can be displayed with the `debug` strategy, but checking the output of the `debug` strategy by hand is error prone. One can also pass an explicitly built input term of the proper structure to the strategy and check whether the strategy still fails or not. Similarly, one can check the result of a strategy by inspecting the output of the `debug` strategy, or by explicit matching:

```
// Explicitly build a term to check a strategy:
!my_term(arg_1,...,arg_n);
s;
// Explicit match to verify that the structure of
// the output term of s is as expected.
?a_term(arg_1,...,arg_j)
```

Unit, component, and integration testing Testing of Stratego programs can be performed using unit, component, and integration tests.

- The Stratego library provides *unit testing* using the `test-suite` strategy contained in the library module `sunit`. This strategy performs a number of tests and produces a test report as output. The `sunit` module contains standard tests for testing that a strategy succeeds for a particular input or that it fails, and for checking the result of a strategy.

  ```
  main = test-suite( !"my-unit-tests",
                     test-1;test-2;test-3)

  test-1 = apply-test(!"test1", id, !"some input")
  test-2 = apply-and-fail( !"test2", fail, !"some input")
  test-3 = apply-and-check( !"test3", sqrt, !4.0, ?2.0)
  ```

  Such test suites can be defined in separate modules as is the case for part of the Stratego library where a module `m-test` defines a test suite for the strategies in the module `m`. One can also define the test suite in the same module and turn it on using the `--main` switch of the Stratego compiler.

  For example, consider the module defining the strategy `some-strategy` and a corresponding test suite below:

  ```
  main = iowrap(some-strategy)

  test-some-strategy = test-suite( .... )
  ```

  Testing of `some-strategy` can be turned on by re-compiling the program with the `--main` switch:

  ```
  sc -i program.r --main test-some-strategy
  ```

- To test a complete Stratego program (*component testing*) it can be passed several fixed input terms, and the output can be matched against corresponding pre-build and verified result terms. In XT we use the test mechanism of Automake [10] which executes a sequence of programs implementing tests and builds a status report. We use `diff` to compare the output of Stratego programs with pre-build terms.

- Most Stratego programs in XT operate on parse-trees or abstract syntax trees and the functionality of these programs is usually small because of the component based approach of XT. Complete programs can be constructed by connecting parsers and pretty-printers to Stratego components, and by combining multiple (Stratego) components together. To test such complex compositions of components, XT defines a collection of *integration tests*. These tests execute the components of XT in different combinations with different inputs and monitor the consistency of XT on a daily basis.

Refactoring When a piece of code from one program is candidate for reuse in another program, it is extracted and put into a separate module. This module is put in a lib directory of GT (GrammarTools). Modules in GT/lib expected to be useful outside GT, can be nominated as candidates for the Stratego library. This way, GT functions as a kind of nursery for the Stratego library.

Adding command line options Command line options and switches are handled automatically when using one of the `iowrap` strategies. By using one of these, the following options/switches are accepted and handled by default:

-b			Write output in binary (BAF) format
-h	-?	–help	Display usage information
-i `<file>`		—input `<file>`	Read input term from `<file>`
-o `<file>`		—output `<file>`	Write output term to `<file>`
-s			Write statistic information after execution
-S		—Silent	Silent execution
-v		—version	Display version information

In addition to these standard command line options, new options can be defined. The `iowrap0` strategy provides a simple mechanism to specify additional switches. Section 3.3.2 contains an example that demonstrates the use of the `iowrap0` strategy and the definition of extra switches. Unfortunately, the generic option handling mechanism of Stratego currently does not support extension of the usage information. Hence, even when new options are accepted, the `--help` switch only displays usage information for the standard set of options.

Optimization The use of standard traversal strategies usually makes programs short and limits language dependence. However, it is often difficult to choose between all different traversals that are provided by Stratego. When several candidate traversal strategies are available to perform a particular job, choosing one or another can influence performance significantly due to their specific traversal pattern.

To control the exact behavior of a traversal an obvious approach is to explicitly program it. This is of course somewhat against the Stratego way of thinking, but when performance really matters it might be an approach to follow.

An alternative approach is to use the *skip* strategies. Ordinary traversal strategies do not distinguish between different nodes and during traversal all nodes are accessed. Usually, a transformation only operates on particular sub trees and only a traversal of these trees would suffice. The *skip* strategies accept an additional argument which determines the trees that should be traversed.

For example, the `asfix-yield` program obtains the yield of an AsFix parse tree which corresponds to the original input string. AsFix is a huge parse tree format and most of the information it contains is not used for this transformation. By traversing only those parts of the tree that contain lexical information, performance is improved significantly. Therefore, `asfix-yield` is implemented by defining a skip strategy for the AsFix constructs that only need partial traversal and we define which sub-trees of these constructs should be traversed:

```
asfix-yield1 =
      leaves(printstring, is-string, skip1)
skip1(x) =
      term(id,id,id,id,id,x,x,x,id)
      + appl(id, id, list(x))
      + list(id, id, list(x))
      + lex(x, id)
```

This `skip1` strategy defines that only three of nine sub-terms of a `term` constructs need to be traversed. Similarly, of an `appl` construct only one of three sub-trees will be traversed.

Connecting components Individual XT components should be designed for reuse according to the component based approach of XT. Consequently, their functionality usually will be restricted. Advanced programs can be constructing be connecting such reusable components together using the ATERMS as exchange format.

Several techniques are used in XT for gluing XT components together. Unix pipes and scripting are used most frequently. Unix scripts handle user interaction (including option handling) and execute all components. Pipes are used to transfer data between components. Component gluing with `make` and Makefiles is another frequently used technique. The individual steps involved in the execution of programs can be expressed clearly in Makefiles as make dependencies. This improves understandability of programs in comparison to Unix scripts. In contrast to Unix pipes, `make` uses files to pass data between components. The combination of dependency checking and data exchange by files enables automatic program optimization by only executing components when a re-computation is required. In contrast to Unix scripting, option handling with `make` is difficult.

A third approach being used in XT is Stratego as scripting language to glue components together. This is a somewhat experimental approach because Stratego does not offer real powerful process strategies yet. It offers the `call` strategy to create a new process and execute an external program, data passing between programs has to be programmed explicitly.

For example, GT defines a strategy `sglr` which uses `call` to execute the parser `sglr`:

```
sglr : (tbl, in, out) -> out
       where
         <call> ("sglr", ["-p", tbl, "-i", in, "-o", out])
```

The terms `in` and `out` denote file names of the input and output terms. A Stratego program is responsible for passing data to `sglr` using files. Execution with input/output redirection is not supported.

The option handling mechanism of Stratego and the ability to modify intermediate trees between execution of components are strong motivations for using Stratego for component gluing.

To simplify tool construction and tool use, XT requires that all components performing input/output accept the `-i`, and `-o` switches to specify input and output, respectively. Furthermore, tools should also handle the `-h` to provide brief usage information to the user.

Documentation General documentation about XT, component descriptions including usage information, and tasks descriptions are created and maintained using Wiki [2], a system for collaborative web development. Wiki makes documentation writing and updating extremely simple and there is no delay between documentation writing and documentation availability.

Stratego offers literate programming [7] facilities which can be used to document Stratego programs. Together with a LaTeX style file, this provides the ability to produce LaTeX documents from your Stratego programs. A literate Stratego program starts with `\literate[<module-name>]`, which indicates the start of an ordinary LaTeX file. Code should be placed between `\begin{code}` and `\end{code}`.

```
\literate{my-stratego}
Here comes a program description in \LaTeX.

\begin{code}
strategies
   main = iowrap(id)
\end{code}

The \LaTeX document continuous here
```

When you include literate programs in you LaTeX document, you should add `\usepackage{lit-style}` to the document preamble. The Stratego parser ignores all text except for the text between `\begin{code}` and `\end{code}`. An increasing number of XT components are developed as literate programs.

Usage information about individual components can be obtained with the `-h` command line switch. Specifying the `-h` switch provides a brief listing of available switches. For most components however, also a separate Wiki page is available, which contains additional usage information. To obtain information about or related to a tool, go to our XT web-site at `http://www.program-transformation.org/xt` and enter the tool name in the search entry at the bottom of the Wiki page. This gives you a list of all Wiki pages where the tool name is used and this is an easy way to find the desired information.

3.5 Conclusions

Given our experiences with Stratego in the context of XT, we will attempt to evaluate the language's strengths and weaknesses. Also, we will suggest some possible improvements.

[2]`http://c2.com/cgi/wiki`

3.5.1 Strengths and weaknesses

Strengths Stratego has proven to be a very powerful implementation language within the context of XT. This is due to various strong features of Stratego.

- *Genericity.* Strategy parameters and Stratego's untypedness allow the development of generic reusable strategies. The standard library of Stratego best demonstrates how this helps to shorten your programs and it demonstrates the development of such strategies.
- *Language independence.* With the generic traversals offered by Stratego your programs only need to depend on those language constructs that require transformation. This simplifies the development of transformations significantly when only a relative small part of a language is affected. It also simplifies maintenance of Stratego code after a language change.
- *Large libraries.* The standard libraries in Stratego's distribution offer an extensive set of reusable generic strategies. Though not all of these strategies are reused as often and with the same ease, they significanlty reduce programming effort.
- *Common exchange format* Stratego supports the ATERMS as exchange format. This is in line with the philosophy of XT, and greatly simplifies component-based development.

Weaknesses We also identified a number of weaknesses of Stratego, and the support that is currently available for it.

- *Performance.* One of the most problematic properties of Stratego is the poor performance of its compiler. This has great impact on the development effort of Stratego programs because the development cycle editing–compiling–testing takes so much time.
- *Static checking.* Due to its untypedness, static checking of Stratego programs is limited and there is no way to see what type of data is expected by strategies. This often results in unexpected behavior of Stratego programs.
- *Conceptual complexity.* Stratego is a difficult language to learn because programming with strategies is a technique that most people are unfamiliar with, because the syntax of the language is uncommon, language documentation is still under development, and because of minimal debugging facilities and poor error reporting. Consequently, it takes quite some time and effort to get used to the language.
- *Code comprehension and reuse.* Lack of a type system and minimal documentation of library strategies hampers reuse. Although there is a huge amount of strategies available for reuse, it is hard to find out whether a strategy with a specific functionality is available and how to use it. Finding out what input a strategy expects and what output it returns are re-occurring problems.

3.5.2 Suggestions for improvement

- *Additional checks and error reports.* Short of a type system for Stratego, and an accompanying type-checker, many additional static checks would be helpful to the programmer. For instance, mistakes in arities of strategies and strategy variables could be caught and reported as such by the compiler, instead of leading to C compilation or linking errors.

- *Number of rewrites.* Although the traversal strategy that is used can have great impact on the performance of Stratego programs, there is no mechanism (except for time measurements) to compare the use of different strategies. Additional statistics, like statistics about the number of rewrites for instance, would be of great help.

- *Globally accessible options.* Code depending on command line switches as specified by the user can occur everywhere in a program. The option handling mechanism of Stratego now requires that the list of options is passed through the program. This mechanism pollutes Stratego programs and may affect many strategy definitions. Making the options globally available would simplify Stratego programs, because it makes accessing the command line switches easier and eliminates the need to pass switches through a program.

- *Documentation.* Reuse of strategies contained in the the Stratego library is hampered because it is often not clear how to use a strategy. Adding examples of strategy usage would improve the reusability of the library. Also, a documentation standard for Stratego strategies is required to indicate the arities of strategy argument, types of input and output terms, and the types of variables. A *reference card* listing the most commonly used strategies, and language constructs would assist the novice Stratego user in learning the language.

References

[1] M. van den Brand, H. Jong, P. Klint, and P. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[2] M. de Jonge. A pretty-printer for every occasion. In I. Ferguson, J. Gray, and L. Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000).* University of Wollongong, Australia, 2000.

[3] M. de Jonge, E. Visser, and J. Visser. XT: a bundle of program transformation tools. In *Language Descriptions, Tools and Applications (LDTA 2001).* Electronic Notes in Theoretical Computer Science, To appear.

[4] M. de Jonge and J. Visser. Grammars as contracts. In *Generative and Component-based Software Engineering (GCSE)*, Lecture Notes in Computer Science (LNCS), Erfurt, Germany, To appear. Springer.

[5] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[6] M. M. I. Herman. GraphXML – An XML-based graph description format. In *Symposium on Graph Drawing (GD 2000)*. Springer, 2000. To appear.

[7] D. E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.

[8] E. Koutsofios. Drawing graphs with *dot*. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, Nov. 1996. This report, and the program, is included in the `graphviz` package, available for non-commercial use at `http://www.research.att.com/sw/tools/graphviz/`.

[9] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. In *Language Descriptions, Tools and Applications (LDTA 2001)*. Electronic Notes in Theoretical Computer Science, To appear.

[10] D. Mackenzie and T. Tromey. Automake. Available from `http://www.gnu.org/`.

[11] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[12] E. Visser. *The Stratego Compiler (version 0.4.22)*. Institute of Information and Computing Sciences, Utrecht, The Netherlands, 2001. Most recent version at `http://www.stratego-language.org/`.

[13] E. Visser. *The Stratego Library (version 0.4.22)*. Institute of Information and Computing Sciences, Utrecht, The Netherlands, 2001. Most recent version at `http://www.stratego-language.org/`.

[14] E. Visser. *The Stratego Tutorial (version 0.4.22)*. Institute of Information and Computing Sciences, Utrecht, The Netherlands, 2001. Most recent version at `http://www.stratego-language.org/`.

Chapter 4

SDL Re-engineering for 5ESS Feature Development

Ronald van Haalen[1]

Abstract Part of the operation of Lucent Technologies 5ESS telephone exchange is described using an SDL dialect (Specification and Description Language). SDL describes the behaviour using statemachines. This description can be a textual or a graphical one, but Lucent only uses the textual description. Commercial tools are not available to interpret the Lucent SDL models, because a proprietary dialect (AT&T SDL) is used. By using Stratego and the XT toolset, a tool has been created that transforms the textual AT&T SDL code into a graphical representation of a statemachine. Stratego was used to collect the statenames and transitions from the textual description. The XT tool set and a graph drawing program were used to visualize the statemachine. The resulting graph shows the transitions between the states. This makes it easier to understand the behaviour of the code, because the generated graphical representation abstracts from the details contained in the textual format.

4.1 Lucent Technologies

Lucent Technologies was split off from AT&T in 1996. Lucent currently has 125,000 employees worldwide as of October 2000 (of which 25% outside the U.S.), with locations in more than 90 countries and territories. The major product areas are:

- Optical networks
- Switching networks
- Wireless

The headquarters of Europe, Middle East and Afrika (EMEA) is located in Hilversum. The mission of Lucent Bell Labs Twente (BLT) is to acquire and

[1]Lucent Technologies, Bell Labs Twente, haalen@lucent.com

maintain expertise on next generation network protocols and middleware platforms for future multimedia services and applications. BLT mainly focusses on domains of network architectures, protocol design and validation, Q.o.S. over packet networks, billing and accounting, security, mobility, etc.

4.2 5ESS Telephone exchange

A large part (about 70%) of all telephone traffic routed in the Netherlands is switched by 5ESS. The 5ESS switch has an availability of 99.999%, which means that on average only in 0.001% of the time it is not functioning correctly (due to hardware or software problems) Currently, about 700 million telephone lines are implemented on this product. The code for the 5ESS switch is for the most part written in C and SDL

4.3 SDL background

About SDL:

- It was initially defined by CCITT Z.100 standard as a formal language for expressing requirements
- It has two representations: a graphical and a textual one
- It is tailored for specification of reactive systems

An SDL program describes a statemachine, with a state, a nextstate and an input which causes the transition. The following example illustrates this:

```
STATE A;
INPUT example;
TASK C-code
NEXTSTATE B;
ENDSTATE A;
```

SDL can be used as:

- a formal specification language for expressing behavior requirements
- an implementation language
- documentation: a protocol between the designers and implementers

4.4 AT&T SDL Legacy

AT&T decided to make its own dialect of SDL, namely AT&T SDL. After some time no tool support was given anymore for AT&T SDL, which lead to a legacy problem: There are currently no tools to view or analyze the AT&T SDL code. Still hundreds of Lucent employees work with AT&T SDL code, adding new features and adapting old ones in the 5ESS code. Tools are needed to provide insight in the SDL models, in order to efficiently:

- Reuse the SDL code

- Change existing code in SDL or C
- Teach the new engineers the existing models in SDL
- Documentate

4.5 First idea

The first idea researched was to use commercial tools like Telelogic TAU, that are based on standard SDL. For this, an AT&T SDL program should be converted to the standard SDL format. This conversion turned out to be expensive and difficult. The commercial tools have many additional analyses not useful for AT&T SDL and too much information of the AT&T SDL code would be lost in the conversion.

4.6 Project goals SDL re-engineering

It was decided to develop a tool in a short time that could give insight into SDL code. The goals for the project were to:

- Investigate possible tooling of SDL
- Have new tools to provide insight in the AT&T-SDL legacy code
- Investigate and obtain expertise in re-engineering methods and tools
- Investigate if this kind of "fast" tool development is useful for Lucent

4.7 Approach

An SDL grammar in YACC was available. In the first part of the project this was re-engineered to an SDF grammar (this was done by Ramin Monajemi (BLT) and Merijn de Jonge(CWI))
The XT tools are then used to:

- parse the SDL program
- pretty print the SDL code

Stratego is used for transformations and information gathering. Graphviz is a graph visualisation program, it is used to visualize the finite statemachine.

4.8 Use of Stratego

Stratego is used to convert different formats and to collect information from the SDL code. The following small programs are written in Stratego:

- collect states: returns the list of states defined in the sdl code
- collect transitions: returns a list containing the beginstate, endstate and the transition name

- transitions2ig: changes a list of transitions to the ig-format
- mk-labels: adds anchors and links to a box expression

4.9 Future Work

There are some ideas about what can be done in the future with the XT tools and Stratego within Lucent:

- The prototype is going to be installed and tested for usefulness
- Extend the current prototype:
  - More information: States and transitions dependencies (Stratego would be used for this)
  - Improve the AT&T SDL grammar and the pretty printer
  - Support for C-code
  - Concat-sdl(program that combines multiple SDL files into one file) with XT and Stratego instead of using Perl
  - GraphXML format instead of ig format as intermediate language between a list of transitions and dot.
- Make a similar tool that can visualize the graphical version of SDL

Whether this future work will actually be done, depends mainly on the evaluation of the SDL tool.

Chapter 5

CobolX: Transformations for Improving COBOL Programs

Hedzer Westra[1]

Abstract There is a huge amount of legacy COBOL-code still being used today, e.g. at bank computer systems. Manual maintenance of this code is too expensive. Therefore, it is desirable to automate all kinds of routine modifications. An important requirement in such modifications is the preservation of comments and the original layout of the program.

CobolX is an environment for implementing transformations on COBOL programs. The environment is built using the Transformation Tools (XT) package, which provides tools for parsing (SDF/SGLR), transformation (Stratego) and pretty printing (GPP).

Since COBOL is a complex language, it is necessary to generate all kinds of tools from the syntax definition, since hand coding would not be feasible. The XT package provides grammar tools for generating signatures and pretty-print tables from syntax definitions. These tools were adapted to support layout preservation.

As a first case study, the picture scaling transformation has been implemented. Picture scaling involves propagating information about the affected variables and then scaling picture declarations and constants.

5.1 Introduction

We study the possibilities of using automatic program transformations to make maintenance of legacy code easier. In particular, transformations on COBOL are considered. We introduce a Y2K-class transformation on programs in this language. In this paper we present the results of implementing such a transformation system using the XT package. The transformation itself is expressed in the Stratego language.

[1]hhwestra@cs.uu.nl

We describe a new layout preservation system, which makes it possible to preserve the original layout while transformating. With this system it is possible to abstract from the layout in the transformation rules and strategies using generated overlays.

5.1.1 Organisation

First the goal of this project will be discussed in section 5.2. In section 5.3 the implemented transformation rule is explained. Section 5.4 discusses the proposed layout preservation system. Then follows a description of the XT tools that we modify for this system, in section 5.5. An example of transforming with layout is then shown in section 5.6 by means of a tiny language, `Micro`. In section 5.7 the details of the transformation pipe that CobolX uses are explained.

This paper ends with a conclusion in section 5.8 and an appendix containing Stratego sources for the layout preservation system and sources for the `Micro` example language.

5.2 Goal

The goal of this project is to implement a transformation on COBOL programs in Stratego. The details of this transformation are discussed in section 5.3. The results of the Stratego implementation are compared with implementations of the same transformation in Perl and ASF+SDF[2]. The Perl project was called Trafo.

The Software Improvement Group (SIG for short) is a CWI spin-off company that, amongst others, improves software by automatic program transformations. One can find them on the web at [2]. The SIG made the Perl and ASF+SDF implementations and is interested in the possibilities that Stratego might offer for program transformations. They have kindly supplied the results of the Perl project, an Sdf2 COBOL grammar and pre/postprocessing utilities. In [1] they explain the specifics of the required transformations.

5.3 The transformation

In the SIG project, there were eight distinct transformation rules to be implemented. Two of those are done automatically, the other 6 are done manually because that proved to be more time-efficient. For the CobolX project currently only one of the two automatic rules is inspected: picture expansion[3].

The picture expansion problem is comparable to the Y2K-problem: in the original specification of the program it was expected that the so-called `PRODKODE` field would never contain a value higher than 99. Time went by, and there became a need for values up to 299. Not only the PRODKODE fields are affected, but also fields that can contain PRODKODE values, e.g. because the value of a PRODKODE field is copied into it.

[2]There has been an attempt to implement the transformation in Haskell but this was not successfull.

[3]Implementation of other rules is planned for the future. This will provide information about how easy it is to add transformation rules once a transformation system is set up.

Input COBOL variable declaration:

```
01 PRODKODE PIC 99.
```

Desired output:

```
01 PRODKODE PIC 999.
```

Figure 5.1: `PRODKODE` picture expansion example

If `PRODKODE` is in the seek set ...

```
MOVE PRODKODE TO NUMBER
```

... `NUMBER` is added to the found set

Figure 5.2: propagation of affected variables

COBOL variables are declared by position, not by primitive type (e.g. integer or character). The change from picture 99 to picture 299 means the program, which consists of 80,000+ lines of code, has to be refactored. Not an easy task to do manually.

An example of a picture expansion is shown in figure 5.1.

5.3.1 Implementation in Stratego

To implement this rule in Stratego, the following transformation is devised:

1. Search for all variables with a specific name. This is called the *seek set.*
2. Propagate all affected variables through the program until a fixpoint is found. The new set of variables is called the *found set.* An example of propagation can be found in figure 5.2.
3. Rebuild the source, replacing the pictures of the found set by ones that can contain the new `PRODKODE` values.

5.4 Layout Preservation

Implementing the expansion rule is not the biggest problem in this project. The real issue is to modify source as least as possible, because it must be possible to manually maintain the resulting COBOL source. This is severely obstructed by dropping all layout and comments.

Three new data types are defined to satisfy the layout preservation requirement. They are: **Layout**, **LList(a)** and **SList(a)**. The signatures can be found in appendix 5.9.1, together with strategies and rules that support transformations with these types.

Layout terms contain the layout that is available in context-free productions. The example in figure 5.3 illustrates this. The overlay is provided for transformation rules that abstract from the layout. These rules are explained further in section 5.6.

Sdf production:

```
context-free syntax
  "if" Expr "then" Statement "else" Statement -> Statement
    {cons("IfThenElse")}
```

Stratego signature without layout:

```
signature
  constructors
    IfThenElse: Expr Statement Statement -> Statement
```

Stratego signature with layout:

```
signature
  constructors
    IfThenElse: Layout Expr Layout Layout Statement
                Layout Layout Statement -> Statement
```

Stratego overlay:

```
overlays
  IfThenElse(e,s1,s2) = IfThenElse(_ L(" "),e, _ L(" "),
                                   _ L(" "),s1,_ L(" "),
                                   _ L(" "),e2)
```

Figure 5.3: How layout is incorporated in signatures

The _ `L(" ")` construction is a special Stratego notation that is called Build-Default. It is named like this because it supplies a default contructor, e.g. `L(" ")` that is called if this strategy is used as a build. For a match the _ is used; a wildcard that matches everything. This means that if the layout overlay is used for a match strategy, all layout will be accepted. For builds, single spaces are created[4].

The data types **LList(a)** and **SList(a)** are defined for lists and separated lists, respectively. A lot of Stratego library strategies defined in e.g. `list-basic.r` are rewritten to support use of the **LList(a)** data type.

5.5 New and adapted XT tools

A lot of XT utilities need changes to enable layout preservation. These changes are discussed below.

5.5.1 `sdfcons`

`Sdfcons` is used to generate constructor names for all Sdf productions in the COBOL grammar. Unfortunately, the XT version of `sdfcons` uses an $O(m * n^2)$ algorithm to make all the constructor names unique. This algorithm is replaced

[4]This might not be the preferred behaviour in all cases. It has been suggested to use e.g. _ `UL`, where UL means Undefined Layout, which can be replaced by real layout in a separate transformation phase.

by an $O(m * n)$ one, using the built-in ATerm hashing function. This reduces the execution time of the process from two hours to five seconds.

5.5.2 sdf2stratego

There are two utilities for converting an Sdf2 grammar to Stratego: `sdf2sig`, which generates signatures and `sdf2stratego`, which generates signatures and AsFix overlays. Because AsFix representations of COBOL programs are huge, an ATerm version is constructed. These ATerms contain not only the abstract syntax of the COBOL programs, but also all of the layout. Special signatures and overlays that make transforming with layout easy are generated by the new version of `sdf2stratego`.

5.5.3 implode-asfix

To create this new type of ATerm, which contains all layout that is in the AsFix tree, `implode-asfix` is adapted. Layout is not filtered out but imploded into the new data type **Layout**. Because normal and separated lists are now interspersed with layout, two other basic types are defined: **LList(a)** and **SList(a)**, respectively.

5.5.4 ppgen2

`Ppgen2` is created to generated pretty print tables that convert the abstract syntax to concrete syntax using the **Layout** elements.

5.5.5 h0box2text

Because the Box expressions that `ast2abox` yields only contain zero-spaced horizontal boxes, due to conserving all layout in the ATerm, using `box2text` is overkill. And, more important, much slower than `h0box2text`, which only concatenates all strings in a Box. It does no actual layouting.

5.5.6 atermdiff

For debugging it is convenient to be able to compare two ATerms without cluttering the screen with Megabytes of data. This program tries to do as much. It is also used in the check phase of the CobolX transformation pipe.

5.5.7 ppconv

Recently a new type of pretty print tables has been introduced. `Ppconv` converts old tables to new ones.

5.5.8 striplayoutaterm

This strips all layout information from an ATerm.

Transforming without layout:

```
rules
  DeBrace: Assign(v,BracedExpr(e)) -> Assign(v,e)
```

Transforming with layout:

```
rules
  DeBrace:: Assign(?v,?BracedExpr(e)) --> Assign(!v,!e)
```

Figure 5.4: A transformation rule without and with layout preservation

5.6 Micro, an example language

Micro is a simple language to study the layout preservation system. All of the sources are available in appendix 5.12.1. With the Sdf definition of Micro it is possible to generate Stratego signatures and overlays and a pretty printing table, using the tools described in section 5.5.

As an example of a transformation that is possible with Micro, we present an expression simplifier. In figure 5.4 two rules are shown. Both do the same task, namely removing braces from an expression. The first cannot handle layout, though, whereas the second can. One can see there are only minimal changes needed to add handling of layout, and that the second rules abstracts from the layout: the generated overlay takes care of that.

Maybe the second rule in figure 5.4 looks unfamiliar. The `rule::  input --> output where strategies` rule is syntactic sugar for `rule = input; where (strategies); output` whereas `rule:  input -> output where strategies` is syntactic sugar for `rule = ?input; where (strategies); !output`.

This little difference means that the left- and righthand side of this special rule should be a strategy in stead of a term, which is needed in this situation to be able to work with the layout overlays.

Two drawbacks of the generated layout overlays are:

- Only transformations of the form shown in figure 5.4 are supported.
- New terms will contain single spaces as layout. If other layout is required for new terms, e.g. for indentation, this will have to be done manually.

An example of an unsupported type of transformation:

```
DoRepeat::
  (Assign(?v,BracedExpr(?e)),?n)
-->
  !Repeat(n,Assign(v,e))
```

This will replace all original layout with spaces. The solution is to explicitly refer to the layout terms:

```
DoRepeat::
  (Assign(?v,?l1,?l2,BracedExpr(?e)),?n)
-->
  !Repeat(n,Assign(v,l1,l2,e))
```

5.7 The CobolX transformation process

The transformation process consists of two phases: initialisation and transformation. In figure 5.5 the architecture of the system is depicted schematically.

5.7.1 Initialisation

The first phase takes as its input the Cobol.2 Sdf grammar and produces:

- the same Sdf grammar annotated with ATerm constructors,
- a Stratego signature and overlays for the layout preservation system and
- a pretty printing table.

The Sdf grammar is used to build a parse table. The Stratego signature and overlays are included when compiling CobolX. Only if the Cobol grammar changes the initialisation process is executed again.

5.7.2 Transforming a file

The second phase is a transformation pipe that is applied for each COBOL file that is transformed. The elements of the pipe are, in input/ouput order:

action	produced data type	file extension
copy input	COBOL source	cbli
preprocess	preprocessed COBOL source	pci
parse	COBOL Binary AsFix tree	pcas
implode	uncompressed COBOL ATerm	pcai
CobolX	uncompressed COBOL ATerm	pcao
pretty printing (ast2abox)	BOX	h0box
pretty printing (h0box2text)	preprocessed COBOL	pco
postprocessing	COBOL source	cblo

When this pipe has finished, some checks are applied to make sure that the output is correct. These checks are:

- Compare the COBOL output with the Trafo output, to see if the transformation rule works identical to Trafo's.
- Compare the input and output COBOL source, to find out what CobolX has changed.
- Reparse the output COBOL and compare the resulting ATerm with the CobolX output, to check that the pretty printing phase worked correctly; pretty printing shouldn't change the semantics. This check proved to be useful for finding incorrectly placed layout, so it also serves as a check for the transformation phase.

5.7.3 Results

The CobolX project is not fully completed at this moment. However, a preliminary overview of the quality and speed of the system can already be made.

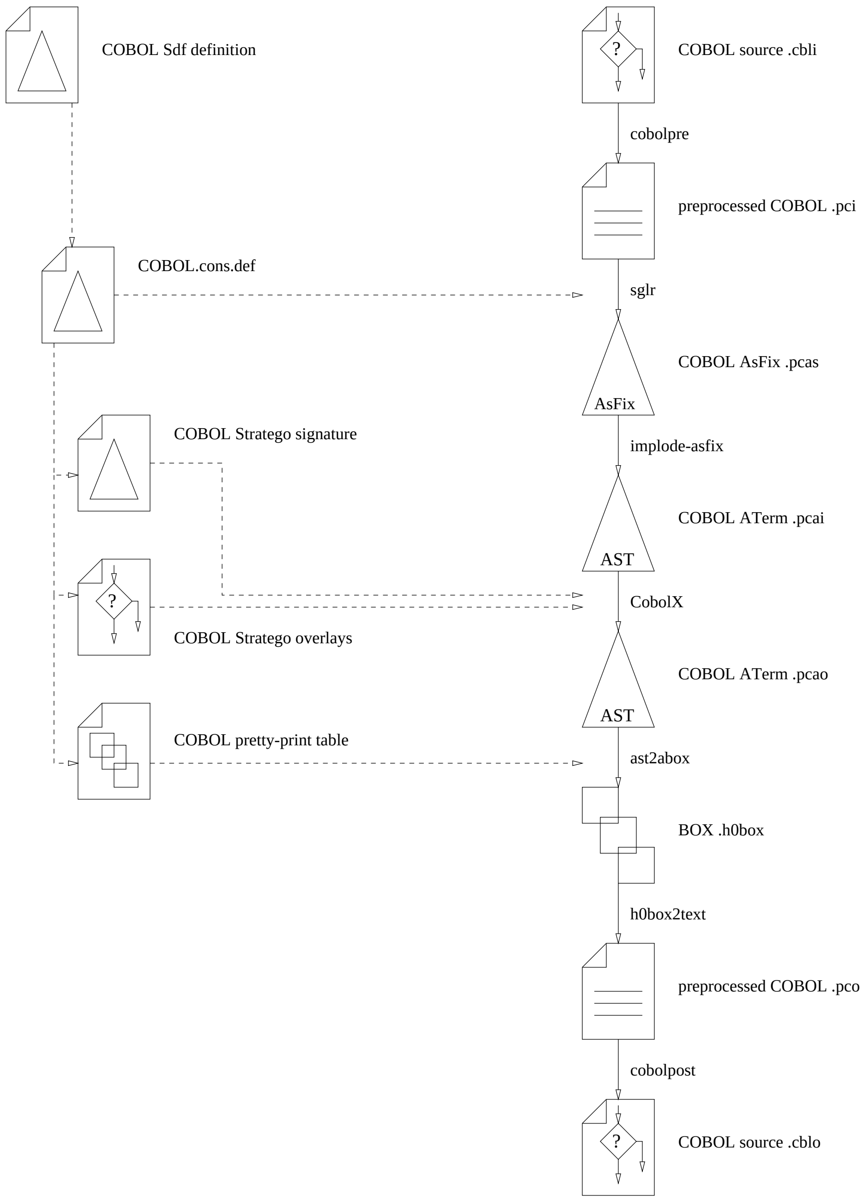

Figure 5.5: CobolX architecture

Trafo vs. CobolX

We compare the results of applying the picture expansion transformation rule to 82 files. For Trafo and CobolX, we obtain the following numbers:

	automatic	manual	total
Trafo	217	28	245
CobolX	161		161
equal	155	5	160
misses	61	23	84
extra	1		1

The Trafo project missed 6 possible automatic changes, 5 of which were done manually. The other seems to have been overlooked.

CobolX misses 23 manual changes, probably because they are only detectable by human inspection. Copybooks, which are comparable to C include files, aren't considered. This means that not all variable declarations are available for the propagation phase.

The 61 automatic changes that CobolX misses are due to a limitation of the implemented propagation rule. This will be fixed in the future. Another fix that is needed is to transform all 85 files. The remaining 3 cannot be transformed by CobolX because of some bugs in the adapted XT tools.

Execution time

It takes about two and a half hours to transform the 82 files on a FreeBSD 4.2 system running on a 500MHz Pentium III with 384 MB memory. In figure 5.6 we plot the execution time (cumulatively subdivided into 4 different phases) against the file size of the input file. Note that the transformation phase takes very little time, in comparison to the parse and check phases. Figure **??** shows that this is due to `implode-asfix`. A possible explanation for the long execution time is that it has to transform AsFix files, which are quite big. In figure **??** one can see that they are indeed the largest files that are handled during the transformation pipe.

5.8 Conclusion

As shown above, it is possible to use Stratego to efficiently transform programs, changing only the parts that need to be changed. Given an Sdf definition one can generate all necessary sources. The transformation itself can then be expressed in Stratego using the generated overlays, which make it possible to abstract from the layout that is contained in the transformed ATerm.

In a best-case scenario, only small modifications will have to be made to add layout preservation to an existing Stratego program. Sometimes, however, the overlays do not suffice. Layout must then be explicitly handled.

The programs mentioned in section 5.5 will be added to the XT distribution in due time.

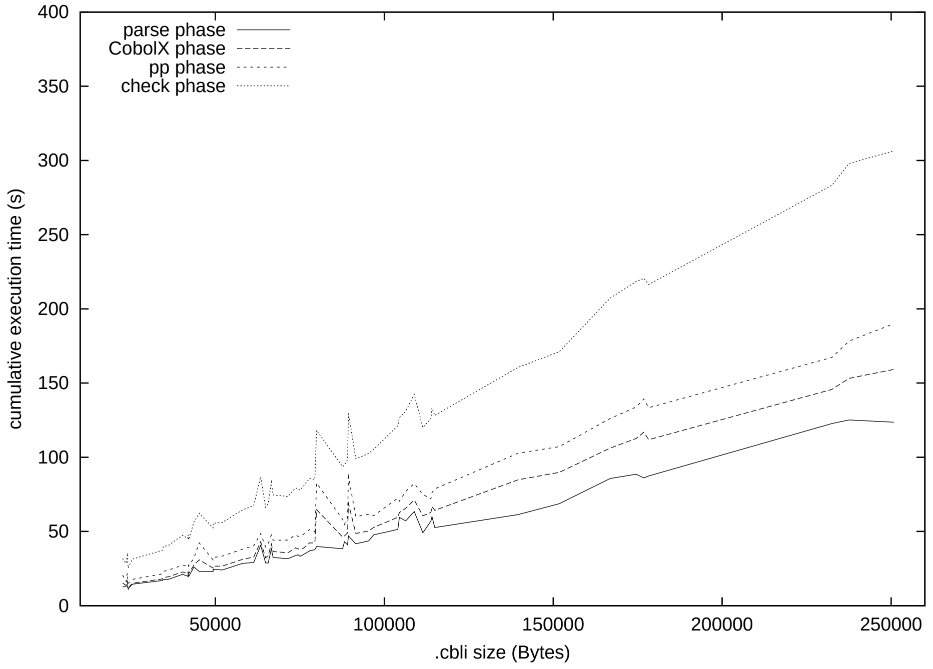

Figure 5.6: Execution time for all transformed files

References

[1] Steven Klusener, Ralf Lämmel, and Chris Verhoef. Going from PIC 99 to PIC 999. Draft, June 2000.

[2] Software Improvement Group website. http://www.software-improvers.com.

5.9 Sources

5.9.1 Library functions

5.10 LayoutPreserve

```
module LayoutPreserve
  signature
    constructors
      NL :           Layout
      L  : String -> Layout
      UL :           Layout
      US :           Layout
      Label: String * s -> Label
      Bracket: String * Layout * s * Layout * String -> Bracket
    overlays
      Bracket(s) = Bracket(_ "(",_ UL,s,_ UL,_ ")")

rules

  layconc: (NL,NL)       -> NL
  layconc: (l@L(_),NL)   -> l
  layconc: (NL,l@L(_))   -> l
  layconc: (L(l1),L(l2)) -> L(<concat-strings>[l1,l2])

  layconc: (UL,NL)       -> UL
  layconc: (NL,UL)       -> UL
  layconc: (UL,UL)       -> UL

// is this the behaviour we want??
  layconc: (l@L(_),UL)   -> l
  layconc: (UL,l@L(_))   -> l

strategies

  layeq(s) = rec r({f,g,fs,gs:
          (?f,?f)
        <+ (isLayout,isLayout)
        <+ \a@(f#(fs),f#(gs)) -> a where
                <eq>(<length>fs,<length>gs);<zip(r)>(fs,gs)\
        <+ \a@(f#(fs),g#(gs)) -> a where <s>(f,g)
//this results in pretty large diffs..
//;<eq>(<length>fs,<length>gs);
//<zip(r)>(fs,gs)
                \
//      <+ echo(!"shouldn't be reached: ");fail
        })

  isLayout = NL + UL + L(id)
```

5.11 LList

```
module LList
imports lib apply
  signature
    constructors
      LCons : a * Layout * LList(a) -> LList(a)
    overlays
      LCons(hd,tl) = LCons(hd,_ UL,tl)
      lsingle(s) = LCons(s,_ UL,Nil)

rules

  ll2singleton: LCons(x,_,[]) -> [x]

strategies

  set-last-layout(s) = rec r(LCons(id,s,[]) <+ LCons(id,r))

  lmap(s) = rec r(LCons(s,r) + Nil)
  lmap''(s) = rec r(Nil <+ apply2(s,LCons(id,r)))
  lmap'(s) = rec r(
   {x',l': \ LCons(x,l,xs) -> LCons(x',l',<r>xs)
        where <s>(x,l) => (x',l') \}
    + Nil)

  lthread-map(s) = rec x((LCons^T(s,id,x) + Nil^T))

  lfilter(s) = rec x([] + (LCons(s,x)<+lTl;x))
  lfetch(s) = rec x(LCons(s,id) <+ LCons(id,x))
  llength = rec x([]; !0 + lTl; x; \n -> <add> (n, 1)\ )

(* LList(List(a)) -> List(a) *)
  lconcat = rec x([] + \ LCons(l,ls) -> <at-end(<x>ls)>l \ )

(* List(LList(a)) -> LList(a) *)
  lconcat' = rec x([] + \ [l|ls] -> <at-l-end(<x>ls)>l \ )
  at-l-end(s) = rec x(LCons(id,x) + [];s)
  at-l-last(s) = rec x(LCons(s,[]) <+ LCons(id,x))
```

```
(* LList(LList(a)) -> LList(a) *)
  lconcat'' = rec x([] + \LCons(l,l1,ls) -> <at-l-end'(<x>ls)>(l,l1)\)

  at-l-end'(s) = ?(l,l1);!l;rec x(   LCons(id,
                                           \ l2 -> <layconc>(l2,l1) \,
                                           ?[];s)
                                  <+ LCons(id,x)
//                                  <+ (?[];s=>ls;!LCons(ls,l1,[]))
)

  ConstoLCons = rec x([]+\[l|ls]->LCons(l,<x>ls)\)
  LConstoCons = rec x([]+\LCons(y,ys)->[y|<x>ys]\)

  lLast = rec r(\LCons(x,Nil) -> x\ + \LCons(_,xs) -> <r>xs\)
  lInit = rec r(\LCons(x,Nil) -> Nil\ + LCons(id,r))

rules

  lTl: LCons(x,xs) -> xs
  lHd: LCons(x,xs) -> x

  lconc: (l1,l2) -> <at-l-end(!l2)> l1
```

5.12 SList

```
module SList
  signature
    constructors
      SCons : a * Layout * String * Layout * SList(a) -> SList(a)
  overlays
    SCons(hd,tl)
      = SCons(hd,_ L(" "),_ ",",_ L(" "),tl)
```

5.12.1 Micro

5.13 micro.def

```
definition

module Main
imports Program Disambiguate Layout AddPrecision

module Program
imports Statement
exports
  sorts Program
  context-free syntax
    Statement+                  -> Program {cons("Program")}

module Statement
imports Expr Var Bool
exports
  context-free syntax
    "if" Bool "then" Statement ("else" Statement)?
                                -> Statement {cons("IfThenElse")}
    "{" Statement+ "}"          -> Statement {cons("Block")}
    "do" Statement "while" Bool -> Statement {cons("DoWhile")}
    Var ":=" Expr               -> Statement {cons("Assign")}

module Expr
imports Number Var
exports
  context-free syntax
    Expr "+" Expr               -> Expr {cons("Plus")}
    Expr "-" Expr               -> Expr {cons("Minus")}
    Expr "/" Expr               -> Expr {cons("Div")}
    Expr "*" Expr               -> Expr {cons("Mul")}
    "(" Expr  ")"               -> Expr {cons("BracedExpr")}
    Number                      -> Expr {cons("NumExpr")}
    Var                         -> Expr {cons("VarExpr")}

module Number
exports
  lexical syntax
    [0-9]+                      -> Number {cons("Number")}

module Bool
imports Expr
exports
  context-free syntax
    "True"                      -> Bool {cons("True")}
    "False"                     -> Bool {cons("False")}
    Expr "=" Expr               -> Bool {cons("Eq")}
    Expr "<" Expr               -> Bool {cons("Lt")}
    Expr ">" Expr               -> Bool {cons("Gt")}
    Expr ">=" Expr              -> Bool {cons("Ge")}
    Expr "<=" Expr              -> Bool {cons("Le")}
    Expr "!=" Expr              -> Bool {cons("Ne")}
    "not" Bool                  -> Bool {cons("Not")}
    Bool "and" Bool             -> Bool {cons("And")}
    Bool "or" Bool              -> Bool {cons("Or")}
    "(" Bool  ")"               -> Bool {cons("BracedBool")}

module Var
exports
  lexical syntax
    [a-zA-Z\_][a-zA-Z0-9\_\-]*  -> Var {cons("Var")}

module Disambiguate
imports Expr
exports
  context-free priorities
    Expr "*" Expr               -> Expr {left}  >
    Expr "+" Expr               -> Expr {left}  >
    Expr "-" Expr               -> Expr {left},

    "not" Bool                  -> Bool {left} >
    Bool "or" Bool              -> Bool {left} >
    Bool "and" Bool             -> Bool {left}

module AddPrecision
imports Number
exports
  context-free syntax
    [0][0-9]+                   -> Number {reject}
    "-" [0]                     -> Number {reject}

module Layout
exports
  lexical syntax
    [\ \t\n]                    -> LAYOUT {cons("LAYOUT")}
  context-free restrictions
    LAYOUT? -/- [\ \t\n]
```

`micro.r`

This file is created by `sdf2stratego` and contains the signatures and overlays for the Micro language.

```
definition

module Main
```

```
  imports LayoutPreserve LList SList Program Disambiguate Layout AddPrecision

module Program
  imports LayoutPreserve LList SList Statement
  signature
    constructors
      Program : LList(Statement) -> Program
      START : Layout * Program * Layout -> START
  overlays
    START(a_0)
      = START(_ L(" "),a_0,_ L(" "))

module Statement
  imports LayoutPreserve LList SList Expr Var Bool
  signature
    constructors
      Block :      Layout * LList(Statement) * Layout -> Statement
      IfThenElse : Layout * Bool * Layout * Layout *
                   Statement * Layout * Option(Statement) -> Statement
      DoWhile :    Layout * Statement * Layout * Layout * Bool -> Statement
      Assign :     Var * Layout * Layout * Expr -> Statement
  overlays
    Block(c_0)
      = Block(_ L(" "),c_0,_ L(" "))
    IfThenElse(d_0,e_0,f_0)
      = IfThenElse(_ L(" "),d_0,_ L(" "),_ L(" "),e_0,_ L(" "),f_0)
    DoWhile(g_0,h_0)
      = DoWhile(_ L(" "),g_0,_ L(" "),_ L(" "),h_0)
    Assign(i_0,j_0)
      = Assign(i_0,_ L(" "),_ L(" "),j_0)

module Expr
  imports LayoutPreserve LList SList Number Var
  signature
    constructors
      Plus : Expr * Layout * Layout * Expr -> Expr
      Minus : Expr * Layout * Layout * Expr -> Expr
      Div : Expr * Layout * Layout * Expr -> Expr
      Mul : Expr * Layout * Layout * Expr -> Expr
      BracedExpr : Layout * Expr * Layout -> Expr
      NumExpr : Number -> Expr
      VarExpr : Var -> Expr
  overlays
    Plus(k_0,l_0)
      = Plus(k_0,_ L(" "),_ L(" "),l_0)
    Minus(m_0,n_0)
      = Minus(m_0,_ L(" "),_ L(" "),n_0)
    Div(o_0,p_0)
      = Div(o_0,_ L(" "),_ L(" "),p_0)
    Mul(q_0,r_0)
```

```
      = Mul(q_0,_ L(" "),_ L(" "),r_0)
    BracedExpr(s_0)
      = BracedExpr(_ L(" "),s_0,_ L(" "))

module Number

module Bool
  imports LayoutPreserve LList SList Expr
  signature
    constructors
      True : Bool
      False : Bool
      Eq : Expr * Layout * Layout * Expr -> Bool
      Lt : Expr * Layout * Layout * Expr -> Bool
      Gt : Expr * Layout * Layout * Expr -> Bool
      Ge : Expr * Layout * Layout * Expr -> Bool
      Le : Expr * Layout * Layout * Expr -> Bool
      Ne : Expr * Layout * Layout * Expr -> Bool
      Not : Layout * Bool -> Bool
      And : Bool * Layout * Layout * Bool -> Bool
      Or : Bool * Layout * Layout * Bool -> Bool
      BracedBool : Layout * Bool * Layout -> Bool
  overlays
    Eq(v_0,w_0)
      = Eq(v_0,_ L(" "),_ L(" "),w_0)
    Lt(x_0,y_0)
      = Lt(x_0,_ L(" "),_ L(" "),y_0)
    Gt(z_0,a_1)
      = Gt(z_0,_ L(" "),_ L(" "),a_1)
    Ge(b_1,c_1)
      = Ge(b_1,_ L(" "),_ L(" "),c_1)
    Le(d_1,e_1)
      = Le(d_1,_ L(" "),_ L(" "),e_1)
    Ne(f_1,g_1)
      = Ne(f_1,_ L(" "),_ L(" "),g_1)
    Not(h_1)
      = Not(_ L(" "),h_1)
    And(i_1,j_1)
      = And(i_1,_ L(" "),_ L(" "),j_1)
    Or(k_1,l_1)
      = Or(k_1,_ L(" "),_ L(" "),l_1)
    BracedBool(m_1)
      = BracedBool(_ L(" "),m_1,_ L(" "))

module Var

module Disambiguate
  imports LayoutPreserve LList SList Expr

module AddPrecision
```

```
  imports LayoutPreserve LList SList Number

module Layout
```

```
micro.pp
```

This file is created by ppgen2 and contains the pretty printing rules for the Micro language.

```
[
   START -- Hhs=0[_1 _2 _3],
   L -- Hhs=0[_1],
   NL -- Hhs=0[],
   SCons -- Hhs=0[_1 _2 _3 _4 _5],
   SNil -- Hhs=0[_1],
   LCons -- Hhs=0[_1 _2 _3],
   LNil -- Hhs=0[_1],
   TCons -- Hhs=0[_1 _2],
   TNil -- Hhs=0[],
   Program -- Hhs=0[_1],
   Block -- Hhs=0[KW["{"] _1 _2 _3 KW["}"]],
   IfThenElse -- Hhs=0[KW["if"] _1 _2 _3 KW["then"] _4 _5 _6 _7],
   IfThenElse.7:seq -- H ["else" _1 ],
   DoWhile -- Hhs=0[KW["do"] _1 _2 _3 KW["while"] _4 _5],
   Assign -- Hhs=0[_1 _2 KW[":="] _3 _4],
   Plus -- Hhs=0[_1 _2 KW["+"] _3 _4],
   Minus -- Hhs=0[_1 _2 KW["-"] _3 _4],
   Div -- Hhs=0[_1 _2 KW["/"] _3 _4],
   Mul -- Hhs=0[_1 _2 KW["*"] _3 _4],
   BracedExpr -- Hhs=0[KW["("] _1 _2 _3 KW[")"]],
   NumExpr -- Hhs=0[_1],
   VarExpr -- Hhs=0[_1],
   True -- Hhs=0[KW["True"]],
   False -- Hhs=0[KW["False"]],
   Eq -- Hhs=0[_1 _2 KW["="] _3 _4],
   Lt -- Hhs=0[_1 _2 KW["<"] _3 _4],
   Gt -- Hhs=0[_1 _2 KW[">"] _3 _4],
   Ge -- Hhs=0[_1 _2 KW[">="] _3 _4],
   Le -- Hhs=0[_1 _2 KW["<="] _3 _4],
   Ne -- Hhs=0[_1 _2 KW["!="] _3 _4],
   Not -- Hhs=0[KW["not"] _1 _2],
   And -- Hhs=0[_1 _2 KW["and"] _3 _4],
   Or -- Hhs=0[_1 _2 KW["or"] _3 _4],
   BracedBool -- Hhs=0[KW["("] _1 _2 _3 KW[")"]]
]
```

5.14 micro-main

```
module test-main
```

```
imports test-simplify lib io Main
strategies

simple=topdown(repeat(SimplifyAssign))
flatten=bottomup(try(DeBlock+DeBrace))

(*
example strategy:
  conv =
  START(Program(?stats)); START(Program(![Assign("i",NumExpr("3"))]))

example rule:
  conv::
    START(Program(?stats))
  -->
    START(Program(![Assign("i",NumExpr("3"))]))
*)

main = stdio(simple;flatten)
```

5.15 micro-simplify

```
module test-simplify
imports io string Main
strategies

  SimplifyDeepExprs = SimplifyExpr + SimplifyBraced

  zeroOrOne(s) = test(s+all(s))

rules

  DeBlock::
    Block(?[e])
  -->
    !e

  DeBrace::
    Assign(?v,?BracedExpr(e))
  -->
    Assign(!v,!e)

  SimplifyAssign::
    Assign(?var,?expr)
  -->
    !Block(b)
    where <SimplifyAllExprs>(var,expr) => b
```

```
SimplifyAllExprs:
  (var,expr)
->
  <SimplifyDeepExprs> (var,expr,v1,v2)
  where <not(zeroOrOne(?NumExpr(_)+?VarExpr(_)+?NL+?L(_)))>expr
      ; new => v1
      ; new => v2

SimplifyExpr:
  (var,expr#([e1,l1,l2,e2]),v1,v2)
->
 LCons(Assign(v1,e1),
  LCons(Assign(v2,e2),
   LCons(Assign(var,
                <mkterm>(expr,[VarExpr(v1),l1,l2,VarExpr(v2)])),
    [])))
  where <?"Plus"+?"Minus"+?"Div"+?"Mul">expr

SimplifyBraced::
  (?var,BracedExpr(?e),?v,id)
-->
  !LCons(Assign(v,e),
    LCons(Assign(var,BracedExpr(VarExpr(v))),
     []))
```

`example input`

This is an example input for the Micro expression simplifier. It simplifies all expressions with more than one arithmetic operator, keeping as much original layout intact as possible. Single spaces are inserted for every newly created grammar element.

The layouting of this input is intentionally non-standard, to show how the layout preservation and generation of new layout works.

```
{ i := t  + 3 *7
b := 4 *2
  f:=0
}
      if a=3 then c:=4
else

c:=8


  if g>3 then d:=5

a  := (4*8)
{f:=0 }
```

`example output`

```
{ { a_0 := t b_0 := 3 *7 i := a_0  + b_0  }
b := 4 *2
  f:=0
}
      if a=3 then c:=4
else

c:=8


  if g>3 then d:=5

{ c_0 := 4*8 a := c_0  }
f:=0
```

Chapter 6

Optimizing Pan programs with Stratego

Arne de Bruijn[1]

Abstract Pan is a language for image manipulation. Originally, Pan programs are written with special constructors in the functional language Haskell. These Haskell expressions are compiled to native code, with heavy use of inlining. A common subexpression elimination phase is used to reduce duplicate code. But the code remains too big, so another way to write Pan programs is proposed, where more control over inlining is possible: Pan code is rewritten in FunTiger, an experimental language, and optimized with Stratego rules.

6.1 Introduction

In various domains execution speed is critical for the application. One of these domains is interactive image manipulation. A difficulty with optimizing programs is maintaining a balance between maintainability and the level of optimization. This is addressed by the Pan language [2], where one can write complex image manipulation programs by composing simpler functions. It is an embedded language, and uses the functional language Haskell as host language. Pan programs consist of special Haskell constructors which generate one big expression for the program. The compiler is an Haskell program which rewrites the expression to C code. There are no declarations in the Pan expression, everything is inlined. To prevent large code size and multiple calculations of the same expressions this inlining is partly undone by a common subexpression elimination (CSE) phase in the compiler.

But the CSE doesn't find common function applications in the code. The approach taken here is to prevent unnecessary inlining in the first place, and leave the CSE as it is. Since the current architecture doesn't allow control over the inlining, another approach was needed. This was found in writing Pan programs in the experimental language Tiger [1]. This language was chosen because of prior experience with an implementation of it in Stratego [3].

[1]arne@knoware.nl

Section 2 goes into more detail about Pan, section 3 is a short introduction to Tiger, section 4 explains the generation of C code from Tiger programs, and section 5 describes future work.

6.2 Pan

In Pan an image is a function that takes an x and y coordinate, and returns a color for the given position. In the simplest case the images are just black and white, and the color is represented by a boolean. For example the following is a Pan program to create an image with a vertical bar:

```
vstrip (x,y) = y > -0.5 && y < 0.5
```

This is Haskell syntax to define a function `vstrip` that takes a pair `(x,y)` and returns the boolean `True` if `y` is between -0.5 and 0.5 and `False` otherwise (in the Haskell version of Pan the syntax is slightly different, to make it possible to compile the expression with the Pan compiler).

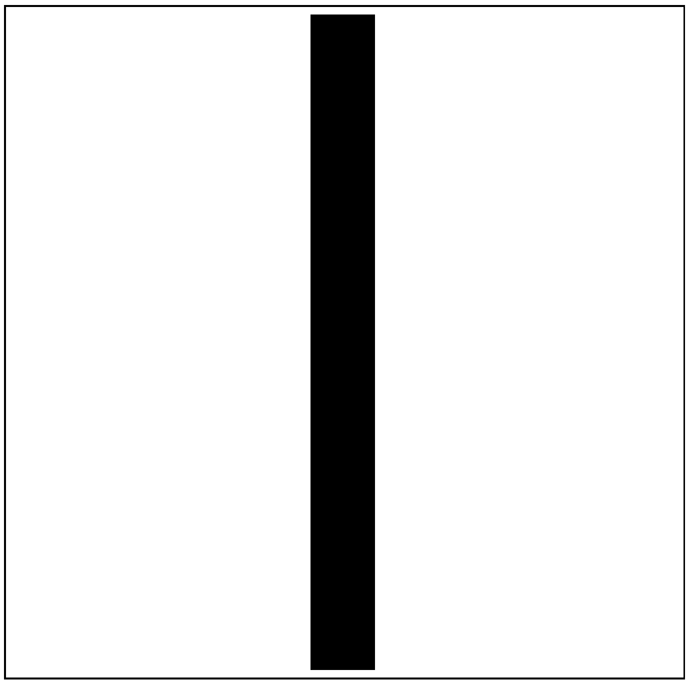

Figure 6.1: vstrip

To display the image, a viewport and a resolution is needed. If the viewport is $(-5,-5)-(5,5)$, and the resolution is 256×256, the following loop renders the image:

```
for y := 0 to 255 do
 for x := 0 to 255 do
  img[y][x] := vstrip(x * 10 / 256 - 5, y * 10 / 256 - 5)
```

To optimize this loop, the function `vstrip` is inlined:

```
for y := 0 to 255 do
 for x := 0 to 255 do
  img[y][x] := (y * 10 / 256 - 5 > -0.5) && (y * 10 / 256 - 5 < 0.5)
```

In this inlined code another optimization becomes possible: since the expression is independent of x, it can be moved out of the inner loop.

```
for y := 0 to 255 do
 let
  a := (y * 10 / 256 - 5 > -0.5) && (y * 10 / 256 - 5 < 0.5)
 in
  for x := 0 to 255 do
   img[y][x] := a
```

This shows the idea of Pan: the image is described in a clean abstract way, but it compiles to specific, optimized code.

6.3 Tiger

The Tiger language is a small Pascal-alike language designed by Andrew W. Appel for his Modern Compiler Implementation series. An example Tiger program to print "Hello World" is the following:

```
let
 function print2(a:string, b:string) = (print(a), print(b))
in
 print2("Hello ", "world");
end
```

In the original implementation of the Tiger compiler in Stratego, the backend generated MIPS assembler code. To make the compiler more flexible it was rewritten to generate C code. The compiler now consists of the following phases:

- Parser
- Desugarer
- Type checker
- Translation to C code
- Pretty printing the C code
- Compiling the C code

The parser takes the source code of a Tiger program and generates a concrete syntax tree. The desugarer rewrites this tree to an abstract syntax tree. The type checker collects all definitions and type declarations in the program, and checks if these are consistently used. It also annotates some declarations with their type, to facilitate further translations. The translation to c code translates the type-checked abstract Tiger syntax to an abstract syntax for C. The pretty printer creates C source code from this abstract syntax. Finally the C code is compiled to an executable.

6.4 Generating C code

A logical choice to generate C code with Stratego is defining an abstract syntax for C code, and writing a pretty printer for this syntax. This way the C code can be generated and manipulated with Stratego rules. The abstract syntax was designed to stay close to the language, to put the intelligence in the generic rewriting rules, and keep the pretty printer as simple as possible.

The abstract syntax has the following elements.

- `CId`: Identifiers
- `CExp`: Expressions
- `CType`: Types
- `CStm`: Statements
- `CFrag`: Fragments (top-level constructs)

With constructors for constants, like:

```
Ccint   : Int -> CExp
Ccstring: String -> CExp
Ccfloat : String -> CExp
```

And for operators, like:

```
Cbinop  : BinOp * CExp * CExp -> CExp
Ccast   : CExp * CType -> CExp
Ccall   : CExp * List(CExp) -> CExp
Cassign : CExp * CExp -> CExp
```

This makes it possible to translate the Tiger abstract syntax to C abstract syntax, for example a function call:

```
TrExp :
  Call(Var(v), args) -> Ccall(Cvar(lab), args')
  where
      <lookup-label> v => (lab, context, _);
      ( // no static link argument for external functions
              <?"external"; !args> context
      <+
              ![<static-links> context | args]
      ) => args'
```

In this example `TrExp` is the rule to translate all expressions. A function call like `print2("Hello ", "world")` is denoted in the abstract syntax as `Call(Var("print2"), [String("Hello "), String("world")])`. This rule first uses `lookup-label` to find the unique label of the function and context where it is defined, and uses that to construct a function call in the C abstract syntax, which would here look like:

```
Ccall(Cvar("g_0print2"),
    [Cvar("_esc"), Ccstring("Hello "), Ccstring("world")])
```

The C variable `_esc` is the static link (it contains a pointer to a struct with the escaping variables of the calling context).

6.5 Future work

The Tiger compiler has almost enough features to write the example Pan programs in Tiger. When this works, the goal is to write an inliner for Tiger, and to find good heuristics that result in fast, but not too large, programs.

References

[1] A.W. Appel, Modern compiler implementation in Java, Cambridge, 1999

[2] The Pan Home Page: `http://research.microsoft.com/~conal/pan/`

[3] Tiger in Stratego: `http://www.cs.uu.nl/~visser/stratego/tiger/`