Preliminary Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications LDTA 2009

Torbjörn Ekman and Jurgen Vinju

27th—28th of March 2009

Preface

These are the preliminary proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009). This year the workshop was expanded from previous years, to a two-day satellite event of ETAPS in York on March 27th and 28th 2009.

LDTA is an application and tool oriented forum on meta programming in a broad sense. A meta program is a program that takes other programs as input or output. The focus of LDTA is on generated or otherwise efficiently implemented meta programs, possibly using high level descriptions of programming languages. Tools and techniques presented at LDTA are usually applicable in the context of "Language Workbenches" or "Meta Programming Systems" or simply as parts of advanced programming environments or IDEs.

LDTA is organized in cooperation ACM Sigplan. We would like to thank the ETAPS organization, the LDTA organizing committee (Adrian Johnstone, Elizabeth Scott), Erik Van Wyk and University of Minnesota Software Engineering Center for making this workshop possible. The LDTA website is hosted by Technische Universiteit Eindhoven. Many thanks go to the program committee for thoroughly reviewing the large number of submissions. This year we received 40 full submissions of which 16 papers were accepted for presentation.

Our invited speaker this year was Don Syme. We would like to thank him for presenting "An Overview of Language Oriented Programming in F# - from ML to Monads to Reflective Schema Compilation" and for his participation in LDTA 2009.

The final proceedings will appear in Electronic Notes in Theoretical Computer Science. All news about LDTA is announced on `http://www.ldta.info`. We hope you will enjoy reading these proceedings.

Best regards,
Torbjörn Ekman and Jurgen Vinju

co-chairs

Program — Saturday

09:00—10:30 First Session: Invited talk

Don Syme. An Overview of Language Oriented Programming in F# - from ML to Monads to Reflective Schema Compilation.

10:30—11:00 Tea break

11:00—12:30 Second Session

12:30—14:00 Lunch break

14:00—15:30 Third Session

15:30—16:00 Tea break

16:00—18:00 Fourth Session

Panel discussion .

18:00 End of first workshop day

19:30—22:30 Workshop dinner Merchant Taylor's Hall

Program — Sunday

09:00—10:30 Fifth Session

10:30—11:00 Tea break

11:00—12:30 Sixth Session

12:30—14:00 Lunch break

14:00—15:30 Seventh Session

15:30—16:00 Tea break, End of workshop

Program Committee

- Jurgen Vinju, CWI, Amsterdam, The Netherlands (co-chair)
- Torbjörn Ekman, Oxford, United Kingdom (co-chair)
- Walid Taha, Rice University, Houston, USA
- Bob Fuhrer, IBM TJ Watson, USA
- Susan Eisenbach, Imperial College, United Kingdom
- Jean-Marie Jacquet, FUNDP, Namur, Belgium
- Sibylle Schuppe, Chalmers, Sweden
- Elizabeth Scott, RHUL, United Kingdom
- Robert Grimm, NYU, USA
- Judith Bishop, Pretoria, South Africa
- Tudor Girba, Univ of Berne, Switzerland
- Marjan Mernik, University of Maribor, Slovenia
- Thomas Dean, Fondazione Bruno Kessler - IRST, Italy
- Martin Bravenboer, Univ. of Oregon, USA
- Pierre-Etienne Moreau, INRIA-LORIA, France
- Gabi Taentzer, Philipps-Universitt Marburg, Germany
- Joao Saraiva, Universidade do Minho, Braga, Portugal
- Tijs van der Storm, CWI, The Netherlands
- Stephen Edwards, Columbia University, USA
- Peter Thiemann, Universitt Freiburg, Germany

Proceedings

Strengthening the Zipper

Tristan O.R. Allwood [1] Susan Eisenbach [2]

Department of Computing
Imperial College
London
United Kingdom

Abstract

The *zipper* is a well known design pattern for providing a cursor-like interface to a data structure. However, the classic treatise by Huet only scratches the surface of some of its potential applications. In this paper we take inspiration from Huet, and describe a library suitable as an underpinning for structured editors. We consider a zipper structure that is suitable for traversing heterogeneous data types, encoding routes to other places in the tree (for bookmark or quick-jump functionality), expressing lexically bound information using contexts, and traversals for rendering a program indicating where the cursor is currently focused.

Key words: zipper, cursor, boilerplate, bookmarks, traversal, generalized algebraic data types,

1 Introduction

We have recently found ourselves implementing an interactive tool for visualising and manipulating an extended λ-calculus, F_C [7], which is the current, explicitly typed, intermediate language used in GHC. Our editor allows the user to view an entire F_C term (e.g., a Haskell function translated to F_C). The user has a cursor, which indicates the current subterm of the entire term that has the focus. The user can then apply operations to the current subterm, such as applying a β-reduction simplification or unfolding a global definition at the first available place within the subterm.

The tool provides a view of the variables that are currently in scope at the cursor location and this information is also provided to the functions that operate on the currently focused subterm. The tool provides several views on the internal data structure, and the user has ways of manipulating it with the output from what is rendered is just a view.

[1] Email: `tristan.allwood@imperial.ac.uk`
[2] Email: `susan.eisenbach@imperial.ac.uk`

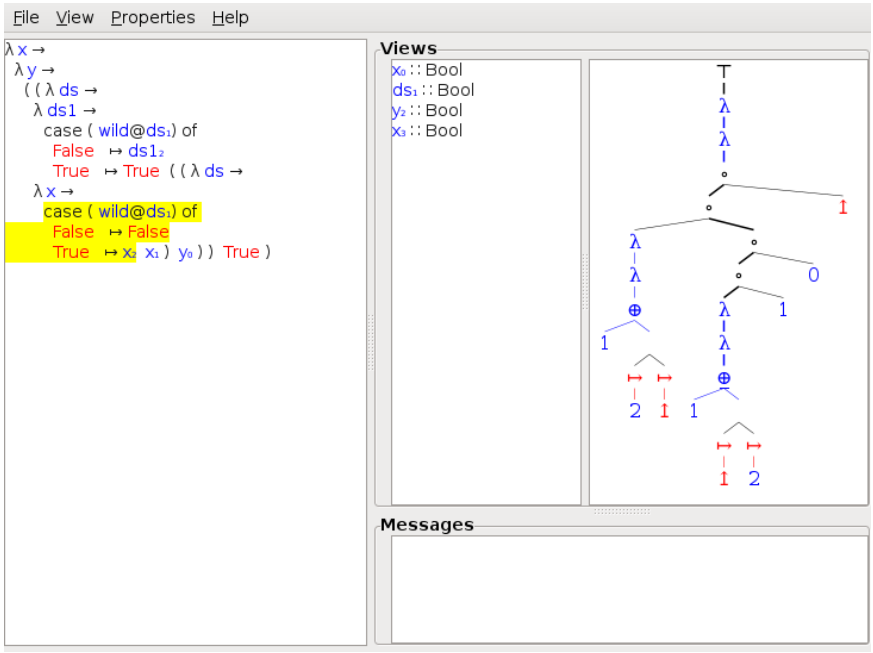


Fig. 1. Our Interactice Core editor.

We wanted to support *bookmarks* i.e., the ability to mark a subterm as *bookmarked* and then later to return the cursor to that location. However, if the user makes a change to the term by manipulating a subterm, we want to be able to detect which bookmarks may have become invalidated by that change, and fix them in some way.

During implementation we developed an internal data structure based around a design pattern similar in many ways to the classic *zipper* [1]. However we had to address several issues that the original paper doesn't cover, and that we couldn't find in the literature. These issues relate to traversal in order to render the different UI views and where complicated by the fact that our underlying term language was made up of several (possibly) mutually recursive data types.

Borne out of our experiences in implementing this tool, we have developed a library, CLASE, that can provide a cursor data structure over a set of data types, and an interface for moving the focus, as-well as traversing the cursor to perform actions such as rendering.

The rest of this paper is organised as follows: We begin in Section 2 by sketching the underlying design of our *Cursor* with a simple example, motivating our use of Generalized Algebraic Data Types (or GADTs, [11]) to allow us to create cursors that can traverse heterogeneous data types. We then introduce CLASE, the cursors it uses and how a user can move a cursor about in

```
data Lam
  = Root Exp
data Exp
  = Abs String Type Exp
  | App Exp Exp
  | Var Integer
data Type
  = Unit
  | Arr Type Type
```

Fig. 2. The Lam Language

Section 3. In Section 4 we motivate some duplication and complexity that can occur when attempting to render a cursor (to e.g., a *String*) and outline how CLASE makes this easier and more idiomatic for a user to do. Another feature of CLASE is its support for bookmarks into the cursor, which are explained int Section 5. Finally in Section 6 we outline related work and conclude.

2 Contexts and simple Cursors

To demonstrate our techniques we have created small language Lam presented in Figure 2. The *Lam* type marks the root of our program, and its sole constructor, *Root*, is a simple wrapper over a Lam expression. [3]

Expressions are either lambda abstractions, *Abs*, which carry a *String* name for their variable, a *Type* for their variable and an expression in which the variable is in scope. Application expressions are the familiar application of two expressions to each other. Variable expressions carry a de Bruijn index [9] indicating which enclosing *Abs* binds the variable this *Var* refers to. Types are either arrow types, *Arr* or some notional unital type, *Unit*.

The following Lam program represents the term $\lambda x :: \tau \to \tau.(x \circ \lambda y :: \tau.(y \circ x))$:

```
Root (Abs "x" (Unit `Arr` Unit) $
        (Var 0) `App` (Abs "y" Unit $ (Var 0) `App` (Var 1)))
```

A simple *Cursor* is a pair of the value (i.e., subterm) currently in focus (sometimes referred to hereafter as *it*), and some *context*, which will allow reconstruction of the entire term. Our cursors are analogous to the zipper design pattern, allowing O(1) movement up or down the tree. In the down direction there are ways to choose which child should be visited.

In Figure 3 we visualise a cursor moving over the Lam value:

[3] Lam refers to the "language", which is defined by the *Lam*, *Exp* and *Type* types.

Root ((*Abs* `"x"` (*Unit* `` `Arr` `` *Unit*) (*Var* 0)) `` `App` `` (*Var* 1))

The cursor in (a) starts focused at the *Root* constructor. Since this constructor is "at the top" of the term's structure, the context is empty. We then in (b) move the cursor down, so the focus is on the *App* constructor, in doing so we add a *Context constructor* (here *ExpToRoot*) to the front of the context. The context constructors explicitly witness the change needed to be applied to the current focus in-order to rebuild one layer of the tree. Moving the focus onto the *Abs* constructor on the left hand side of the application in (c) pushes a new constructor onto the context. *ExpToApp0* both indicates the focus is in the first *Exp* child of an *App* node, and carries the values that where the right-hand-children of the *App* node (*Var* 1). Moving down once more in (d) puts the focus on the *Arr* constructor inside the *Type* child of the *Abs* node, and again pushes an appropriate context constructor to be able to rebuild the *Abs* node should the user wish to move up.

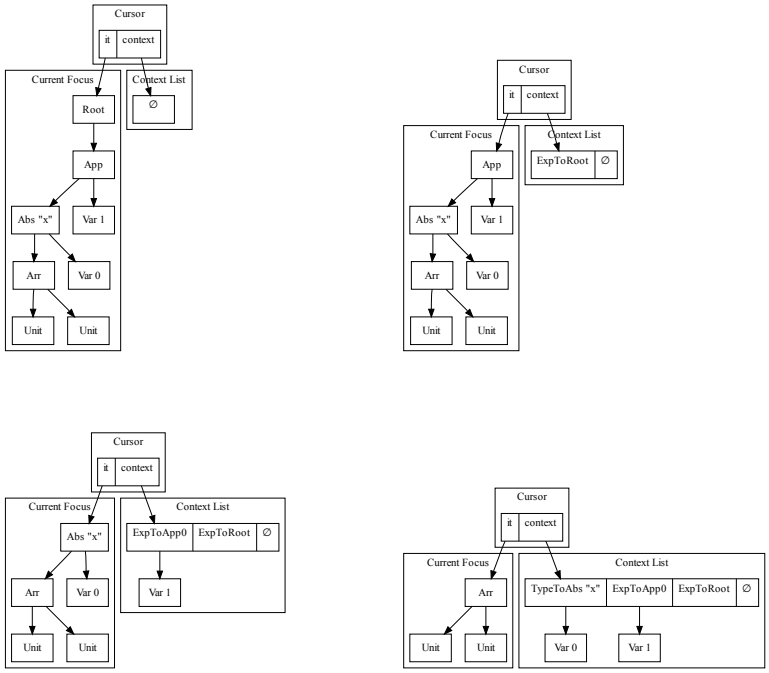


Fig. 3. Cursor structure changes due to moving the focus

In order to be able to move around easily, given our original data types (*Lam*, *Exp* and *Type*), we need context constructors to represent all of the possible contexts for our cursor. We use GADTs to push into the type system

the types that our context constructors expect for their "missing value", and the type of the constructor they ultimately represent. We will later use this extra information to help maintain some general invariants and as a sanity check that our implementation of a cursor is correct.

```
data Context from to where
  ExpToRoot :: Context Exp Lam
  ExpToApp0 :: Exp → Context Exp Exp
  TypeToAbs :: String → Exp → Context Type Exp
  ...
```

As we saw in the diagram, when we move around our term, we build up a stack of *Context*s. If our contexts were ordinary data types we could use a list, however we need to ensure that the *to* parameter of our first *Context* matches up with the *from* parameter of the next *Context*. To do this we use a new data type called *Path*.

```
data Path ctr from to where
  Stop :: Path ctr anywhere anywhere
  Step :: (..) ⇒ ctr from middle → Path ctr middle to → Path ctr from to
```

Stop is akin to the nil, [], at the end of a list, and *Step* is akin to cons, :. Since the intermediate location, *middle*, in *Step* is existentially quantified, we need to provide a way of extracting its type at a later time, and hence the (for the moment unspecified) class constraint.

Our basic cursor structure is then simply:

```
data Cursor here = Cursor{
  it :: here,
  context :: Path Context here Lam
}
```

The current point of focus is denoted by *it*, and the *context* we are in is a path from *here* up to the root of our language, *Lam*. The cursor data structure in our library CLASE extends this data type in two useful ways.

3 CLASE

We now present our library, CLASE, which is a *Cursor Library for A Structured Editor*.[4] As shown in Figure 4, a typical use of CLASE consists of three parts. There is the code the developer has to write, typically the data types that express the language to be structurally edited (e.g., the Lam datatypes), a couple of specific type classes useful for rendering as discussed in Section 4, and of course, the application that uses CLASE itself. CLASE then provides

[4] It is available for download with documentation from [8].

some Template Haskell scripts that automatically generate some boilerplate code, and an API of general functions for building, moving and using CLASE cursors.

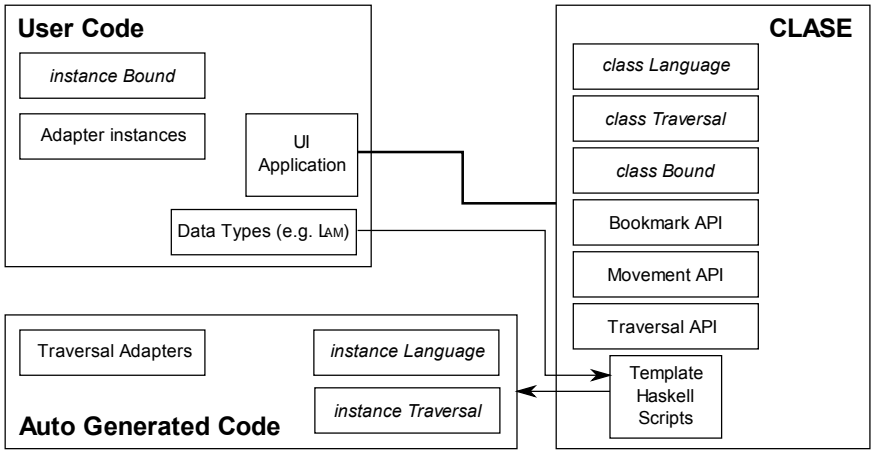


Fig. 4. An overview of using CLASE

CLASE requires the user to implement a typeclass called *Language*, shown in Figure 5. The instance should be the type that is at the root. So for our Lam example, we will make an instance *Language Lam*.

A *Language* needs to have data types (expressed using associated data type families [12]) corresponding to its *Context*s, primitive *Movement*s and a way of reflecting on the different types in the language (*TypeRep*). The *Context Lam* is as shown earlier. Primitive *Movement*s are GADT constructors that witness all of the one-step movements *Up* or *Down* that the cursor could do. For example, for Lam they would be:

```
data Movement d a b where
  MUp :: Movement Down b a → Movement Up a b
  MRootToExp :: Movement Down Lam Exp
  MAppToExp0 :: Movement Down Exp Exp
  MAbsToType :: Movement Down Exp Type
  ...
```

The way of reflecting on the different types in the language is provided by a simple GADT:

```
data TypeRep a where
  ExpT  :: TypeRep Exp
  LamT :: TypeRep Lam
  TypeT :: TypeRep Type
```

```
class Language l where
  data Context l :: * → * → *
  data Movement l :: * → * → * → * → *
  data TypeRep l :: * → *
  buildOne :: Context l a b → a → b
  unbuildOne :: Movement l Down a b → a → Maybe (Context l b a, b)
  invertMovement :: Movement l d a b → Movement l (Invert d) b a
  movementEq :: Movement l d a b → Movement l d a c → Maybe (TyEq b c)
  reifyDirection :: Movement l d a b → DirectionT d
  contextToMovement :: Context l a b → Movement l Up a b
  downMoves :: TypeRep l a → [ExistsR l (Movement l Down a)]
  moveLeft :: Movement l Down a x → Maybe (ExistsR l (Movement l Down a))
  moveRight :: Movement l Down a x → Maybe (ExistsR l (Movement l Down a))
class Reify l a where
  reify :: a → TypeRep l a
data ExistsR l (r :: * → *) where
  ExistsR :: (Reify l a) ⇒ r a → ExistsR l r
data Up
data Down
type family Invert d :: *
type instance Invert Up = Down
type instance Invert Down = Up
data DirectionT a where
  UpT :: DirectionT Up
  DownT :: DirectionT Down
data TyEq a b where
  Eq :: TyEq a a
```

Fig. 5. The *Language* typeclass and supporting data structures and types

This is then linked by the user to the type class *Reify*, which provides a sometimes more convenient way of passing around the witnesses. *ExistsR* is a data type that provides an existential wrapper for something that is parameterised by some type in the language.

With these data types for the language declared, the user then has to provide some primitive functions that rebuild values given a *Context* and a value for it's "hole" (*buildOne*), or take apart a value given an indication of which child we want to become the "hole" (*unbuildOne*).

We also require that all movements are invertible (*invertMovement*), and can be tested for equality (*movementEq*). Movement equality needs to provide a type equality witness in the case that the two movements are the same. All *Movement*s can only move up or down, which is what *reifyDirection* requires,

and all *Context*s must correspond to an upward movement (*contextToMovement*).

Finally, in order to provide more generic move up/down/left/right operations that don't need an explicit witness, the user needs to enumerate all the possible down movements starting from a particular type (*downMoves*) and finally how to logically shift left or right a downward movement (*moveLeft* and *moveRight*). For example *moveLeft MAbsToExp* = *Just* (*ExistsR MAbsToType*)

For simple languages, such as our Lam language, the instance of *Language* is straightforward and easily mechanically derivable. For such languages we provide a Template Haskell function that can automatically generate a module containing the *Language Lam* instance. As the current state of Template Haskell precludes the inline generation of GADTs and Associated Data Types, our generation function outputs the module as a source file that can be imported later.

The Template Haskell script provided with CLASE requires the user to specify the module to create the instance of *Language* in, the root data type (*Lam*), and the types that the cursor should be able to navigate between (*Lam*, *Exp* and *Type*). The library user invokes the script using a splice, $(...), and refers to the root and navigable types using TH quasiquotes (''), as shown:

```
$(languageGen ["Lam", "Language"]
              '' Lam
              ['' Lam, '' Exp, '' Type])
```

The *languageGen* function works by using TH *reify* to access the shape of *Lam*, *Exp* and *Type* data types, and then processes that to work out the simple *Context*s, and from there simple *Movement*s and the implementations of the primitive functions such as *buildOne* and *unbuildOne*.

With a suitable instantiation of *Language*, CLASE then provides the library functions and data types for representing and manipulating cursors. The core *Cursor* data structure is an extended version of the *Cursor* seen previously. It is now parameterised by three types; l is the same type used to instantiate the *Language* typeclass this *Cursor* is used for, x is used for bookmark like behaviour, and will be discussed in Section 5, and a is the type of the current focus. *it* and *ctx* are the current focus and context as before, and for bookmark behaviour we also add a *log* field, that again will be discussed in Section 5.

```
data Cursor l x a = (Reify l a) ⇒ Cursor{
  it  :: a,
  ctx :: Path l (Context l) a l,
  log :: Route l a x
}
```

CLASE provides two ways of moving the focus of a cursor around. The first is a way of applying the specific movements described in the *Language* instance. This is through a function *applyMovement*:

$$\begin{aligned}&\mathit{applyMovement} :: (\mathit{Language}\ l, \mathit{Reify}\ l\ a, \mathit{Reify}\ l\ b) \Rightarrow \\ &\quad \mathit{Movement}\ l\ d\ a\ b \rightarrow \mathit{Cursor}\ l\ x\ a \rightarrow \mathit{Maybe}\ (\mathit{Cursor}\ l\ x\ b)\end{aligned}$$

so given a *Movement* going from a to b (in either direction), and a *Cursor* focused on an a, you will get back a *Cursor* focused on a b. This is a very useful function if a GUI wants to apply a set of movements around a known tree structure. However it does require knowing up-front the type of the *Cursor*'s current focus and that you have a *Movement* which matches it.

CLASE also provides a set of generalized movement operators. These do not need the calling context to know anything about the *Cursor* they are being applied to. There are four ways of generically moving around a tree, up, depth-first down, or moving left/right to the adjacent sibling of where you currently are. Since it is unknown what type the focus of the *Cursor* will be after applying one of these operations, they return *Cursor*s with the type of *it* existentially quantified.

$$\begin{aligned}&\textbf{data}\ \mathit{CursorWithMovement}\ l\ d\ x\ \mathit{from}\ \textbf{where} \\ &\quad \mathit{CWM} :: (\mathit{Reify}\ l\ \mathit{to}) \Rightarrow \mathit{Cursor}\ l\ x\ \mathit{to} \rightarrow \\ &\qquad\qquad \mathit{Movement}\ l\ d\ \mathit{from}\ \mathit{to} \rightarrow \mathit{CursorWithMovement}\ l\ d\ x\ \mathit{from} \\ &\mathit{genericMoveUp} \quad :: (\mathit{Language}\ l) \Rightarrow \\ &\qquad\qquad \mathit{Cursor}\ l\ x\ a \rightarrow \mathit{Maybe}\ (\mathit{CursorWithMovement}\ l\ \mathit{Up}\ x\ a) \\ &\mathit{genericMoveDown} :: (\mathit{Language}\ l) \Rightarrow \mathit{Cursor}\ l\ x\ a \rightarrow \\ &\qquad\qquad \mathit{Maybe}\ (\mathit{CursorWithMovement}\ l\ \mathit{Down}\ x\ a) \\ &\mathit{genericMoveLeft} \quad :: (\mathit{Language}\ l) \Rightarrow \mathit{Cursor}\ l\ x\ a \rightarrow \\ &\qquad\qquad \mathit{Maybe}\ (\mathit{ExistsR}\ l\ (\mathit{Cursor}\ l\ x)) \\ &\mathit{genericMoveRight} :: (\mathit{Language}\ l) \Rightarrow \mathit{Cursor}\ l\ x\ a \rightarrow \\ &\qquad\qquad \mathit{Maybe}\ (\mathit{ExistsR}\ l\ (\mathit{Cursor}\ l\ x))\end{aligned}$$

In the case of *genericMoveUp / Down* we also return the *Movement* constructor that could have been used via an *applyMovement* call to achieve the same effect, should an application find that useful.

4 Rendering and binding

One of the things you want to do with a structured editor is to display the contents, indicating where the current focus is. Assuming you wish to provide a textual rendering, given one of our *Cursor*s, this would require you to:

(i) Convert the value of the focus (i.e., a *Lam*, *Exp* or *Type*) to a *String*.

(ii) Modify the *String* value from rendering the focus to indicate the current focus (e.g., by wrapping it in marking brackets ala `"> ... <"`).

(iii) Render all of the context constructors, passing in the *String* from rendering what is below that constructor as the *String* to use as the value for the "hole" in the constructor.

```
renderExp :: Exp → M String
renderExp (Abs str ty exp) = do
  tys ← renderType typ
  rhs ← addBinding str (renderExp exp)
  return ("λ " ++ str ++ ": " ++ tys ++ " . " ++ rhs)
...
renderCtx :: Context Lam from to → M String → M String
renderCtx (TypeToAbs str exp) rec = do
  tys ← rec
  rhs ← addBinding str (renderExp exp)
  return ("λ " ++ str ++ ": " ++ tys ++ " . " ++ rhs)
renderCtx (ExpToAbs str ty) rec = do
  tys ← renderType ty
  rhs ← addBinding str rec
  return ("λ " ++ str ++ ": " ++ tys ++ " . " ++ rhs)
...
addBinding :: String → M a → M a
...
```

Fig. 6. Parts of a first attempt at rendering a Lam cursor

In Figure 6 we outline part of our first attempt at just rendering expressions and contexts, ignoring the control flow needed to fold context results into each other. Since the Lam language has bound variables, and we wish to render their names in variable position, we perform the computation in some fictional monad M and use its API function *addBinding* to make a binding for a new variable available in a sub-computation.

As is clear in the example, the code is highly repetitious, and the logic for when to call *addBinding* is intermingled with the code for rendering and traversing. In practice we have found this binding code hard to manage (particularly if the language grows to more complexity), and the duplication highly undesirable.

In CLASE we solve this problem by completely factoring away the binding code, and then provide adapters to make writing the rendering code more factored. CLASE has a typeclass *Bound* which allows a user to use the context-constructors generated for their language to express binding constraints. The intuition is that when a traversal moves down through the tree into the "hole" described by the constructor, the user can alter the result of the traversal value.

```
class (Language l) ⇒ Bound l t where
  bindingHook :: Context l from to → t → t
```

For the Lam language, an appropriate instance for our rendering code above would be to wrap the *hole* value in an *addBinding* application whenever we move through an *Abs* from it's *Exp* child, and in all other cases, to leave it unchanged.

```
instance Bound Lam (M a) where
  bindingHook (ExpToAbs str _) hole = addBinding str hole
  bindingHook _ hole = hole
```

CLASE also provides a combinator for doing render-like traversals over a *Cursor*. It is a library function with the signature:

$$completeTraversal :: \forall\ l\ t\ x\ a.(Traversal\ l\ t) \Rightarrow Cursor\ l\ x\ a \rightarrow t$$

The *Traversal* type-class contains functions for performing the three actions itemized above, and the library contains the glue-code to make the traversal work. However instead of requiring the user to implement this type-class directly, CLASE features another Template Haskell script to automatically create an instance of *Traversal* given instances of some adapters. Our TH script (*adapterGen*) generates the follow code for Lam.

```
class LamTraversalAdapterExp t where
  visitAbs :: Exp → t → t → t
  visitApp :: Exp → t → t → t
  visitVar :: Exp → t
class LamTraversalAdapterLam t where
  visitLam :: Lam → t → t
class LamTraversalAdapterType t where
  visitUnit :: Type → t
  visitArr :: Type → t → t → t
class LamTraversalAdapterCursor t where
  visitCursor :: Lam → t → t
instance (LamTraversalAdapterLam t,
  LamTraversalAdapterExp t,
  LamTraversalAdapterType t,
  LamTraversalAdapterCursor t,
  Bound Lam t) ⇒ Traversal Lam t where
  ...
```

As an example, the user logic for rendering an abstraction constructor is now restricted to just the instance of the *visitAbs* function in *LamTraversalAdapter*:

```
instance LamTraversalAdapterExp (M String) where
  visitAbs (Abs str _ _) ty exp = do
    tys ← ty
```

```
    exps ← exp
    return ("λ " ++ str ++ " : " ++ tys ++ " . " ++ exps)
...
```

Using the API is straightforward, given user instances of the adapters above, then converting a *Cursor* to an *M String* is simply a case of calling *completeTraversal* in the correct-type context:

```
render :: Cursor Lam x a → M String
render = completeTraversal
```

5 Routes and Bookmarks

An editor developed using CLASE may want to keep track of multiple locations in the tree (e.g., to provide bookmark or quick-jump functionality). Ideally we would like these bookmarks to be persistent across updates to the tree, and where this is not possible, for there to be some way of dealing with the now invalidated bookmarks.

Any position in the tree can be reached from any other by a series of *Up* movements, followed by a series of *Down* movements. Using the *Path* data type from earlier, we can encode these routes in a new CLASE data type *Route*:

```
data Route l from to where
  Route :: (Reify l mid) ⇒
    Path l (Movement l Up) from mid →
    Path l (Movement l Down) mid to → Route l from to
```

These can be made into a unique routes by disallowing the last *Up* movement to be the inverse of the first *Down* movement, i.e., the following invariant is maintained:

```
route_invariant :: (Language l) ⇒ Route l from to → Bool
route_invariant (Route (Step mup Stop) (Step mdown _))
  = (¬ ∘ isJust) (invertMovement mup `movementEq` mdown)
route_invariant (Route (Step _ ups) downs)
  = route_invariant (Route ups downs)
route_invariant (Route Stop _) = True
```

The CLASE cursor keeps track of a single *Route* to some marked location. We provide an API for extending the current route by a single movement, resetting it, joining two routes together and making a *Cursor* follow a *Route*.

```
updateRoute :: (Language l, Reify l a, Reify l b) ⇒
  Movement l d a b → Route l a c → Route l b c
```

```
resetLog :: Cursor l x a → Cursor l a a
appendRoute :: (Language l, Reify l a,
                Reify l b, Reify l c) ⇒
  Route l a b → Route l b c → Route l a c
followRoute :: (Language l) ⇒
  Cursor l x a → Route l a c → Maybe (Cursor l x c)
```

Should a user application want to bookmark multiple different subterms, this API makes this straightforward to do, and helps ensure the application doesn't forget to update the bookmarks. The user application would have as its state a Cursor with an empty *log*, and a map of integers to routes that lead from the cursor's current location to somewhere else. For example, a Lam GUI may use the following:

```
data CursorHolder where
  CH :: Cursor Lam a a → Map Int (ExistsR Lam (Route Lam a)) →
                         CursorHolder Lam
```

Creating a new bookmark at the current location is just a case of inserting a value of *ExistsR* (*Route Stop Stop*) into the *Map* at the bookmark's key.

When the GUI tries to move the cursor, the main loop would respond to a keypress. We proceed by unwrapping the cursor holder and applying *genericMoveDown* to the cursor.

```
keypressDown :: CursorHolder → CursorHolder
keypressDown ch@(CH cursor@Cursor{}bookmarks) = fromMaybe ch $ do
  (CWM cursor' _) ← genericMoveDown cursor
  ...
```

At this point of the code, the type of *cursor* is $\exists\ a.Cursor\ Lam\ a\ a$ and *cursor*′ is $\exists\ b.Cursor\ Lam\ a\ b$, i.e., the *log* field of *cursor*′ gives a route back to the a. If we attempted to return a new *CursorHolder* containing *cursor*′ (or *resetLog cursor*′) and the original *bookmarks* it would be a type error. The type system enforces that we update all the bookmarks to make the type parameters match up with the new cursor parameters...

```
  ...
  let bookmarks' = Map.map (λ(ExistsR bm) →
                              ExistsR ((log cursor') `appendRoute` bm))
                           bookmarks
  let cursor'' = resetLog cursor'
  return $ CWM cursor'' bookmarks'
```

Jumping to a bookmark is then a case of using CLASE's *followRoute* to update the cursor's location, and then using the same logic as above to up-

date all the bookmarks (including the one that was just followed) to the new location.

Detecting whether a change to the current focused subterm may invalidate a bookmark is also easy. A route will only point inside the current subterm if it has no up components, i.e., it has the shape *Route Stop something*.

6 Conclusions

There has been a lot of discussion about zipper data structures in the Haskell community recently. Practical, popular applications [5] and general libraries [4] are emerging based on the underlying ideas of the original paper [1]. Like our library, these examples take the general principles of contexts and a focal point, and tailor them to specific domains (managing stacks of windows for a window manager, or providing a usable interface for editing a large number of related items, with the option of changing your mind).

One of the fundamental underpinnings of our work (and much of the related work) is that of a one-holed context. These have been discussed in [10], and provide an interesting relationship between differential mathematics and data structures. Indeed it is due to this link that we know we can automatically generate the *Context*s for simple data structures using our Template Haskell scripts.

There are existing reusable, zipper-based libraries in the literature. In [3] the authors consider a data structure that is parametric over the type being traversed, and requires much less boilerplate to implement. However their library does not consider traversals over a heterogeneous data type and there does not appear to be a succinct extension to the work that would allow such a traversal.

In [2] the author presents an elegant GADT based zipper library that is able to traverse across heterogeneous data types, and requires no boilerplate to use. However we believe that it is not a practically useful library without some additional boilerplate being written; the implementation requires that at all use sites a lot of type information is available to allow up/left/right movements, and down movements require the precise type of what is being moved into to be available. In an application that is interactively allowing a user to update the cursor's position, it would require a complicated existential context with type classes or type witnesses being present, to allow these movements to happen. With our library, we provide both the type specific movements, but have also provided the additional boilerplate needed to recover the generic movements that can move a cursor without any additional type constraints being present.

An alternate approach to the cursor library was explored in [6]. Here, the zipper library is parameterised by a traversal function and uses delimited continuations to move around the tree. The authors also show how to support a statically known number of sub-cursors, allowing something like

our route/bookmark functions. They however, are working in the context of filesystems and do not need to consider lexically bound information in the interface they present.

Unsurprisingly, there is always more functionality we could add to our library. We have also only looked so far at *Language* instance generation for simple languages, we have not considered cursors for languages that are themselves parameterised by types, or languages with GADTs in them, both of these could present interesting challenges for auto-generating their *Language* instance.

Furthermore, the zipper data structure was originally designed around the idea of needing to perform local updates and edits, and not necessarily global traversals; while we justify this by arguing that in an editor context many local edits and changes may take place between the global renders; we should perform some performance and complexity analysis of our global traversals against some alternative schemes.

There are some other issues; we are using some experimental features of GHC (e.g., type families), which are not completely implemented yet - when a complete implementation is released, we will be able to neaten up some of the automatically generated instances. Also, Template Haskell does not support the generation of GADTs or type family instances and so our generation scripts output the source code for compilation to new files; this is an ugly indirection step that we would like to avoid in future.

CLASE was borne out of experience of implementing a structured editor. We now want to retrofit CLASE back into our structured editor, and in doing so hopefully find other useful traversals and features we can generalise out that may be useful in a more general setting.

We have outlined a cursor library based on ideas from Huet's original paper, but using GADTs to allow navigation around a heterogeneous data type. We believe that cross-datatype zippers are a useful extension to the original zipper. The provision of the generic moveUp/moveDown/moveLeft/moveRight API across these heterogeneous data type was the major difficulty.

Using CLASE makes it straightforward to render a cursor, and encode bookmarks.

The code presented has been split into three parts, that which the user provides, that which forms a generic library, and that which we automatically generate using Template Haskell. At no point has the user been required to implement any boiler-plate code themselves.

Acknowledgements

We would like to thank the many people who have commented on tool demos and talks on this work at Anglo Haskell and the Haskell Symposium. Many thanks also to the Slurp research group at Imperial, for their good ideas and useful ways to present our work.

References

[1] Huet, G. The zipper. Journal of Functional Programming, 7(5):549-554, 1997

[2] Adams, M. Functional Pearl: Scrap Your Zippers. Unpublished, 2007

[3] Hinze, R. and Jeuring, J. Functional Pearl: Weaving a Web in J. Functional Programming, 11(6):681-689, November 2001.

[4] Yorgey, B. zipedit library, 2008, http://byorgey.wordpress.com/2008/06/21/zipedit/

[5] Stewart, D. Roll Your Own Window Manager: Tracking Focus with a Zipper (Online), 2007, http://cgi.cse.unsw.edu.au/ dons/blog/2007/05/17

[6] Kiselyov, O. Tool demonstration: A zipper based file/operating system. In Haskell Workshop. ACM Press, September 2005

[7] Sulzmann, M. and Chakravarty, M. M. T. and Jones, S. P. and Donnelly, K. System F with Type Equality Coercions, in The Third ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'07), January 2007.

[8] Allwood, T. Clase library download and screenshots, (Online), 2008, http://www.zonetora.co.uk/clase/.

[9] de Bruijn, N. G. Lambda calculus notation with nameless dummies. a tool for automatic formula manipulation with application to the church-rosser theorem, in Indagationes Mathematicae (34) 381–392, 1972

[10] McBride, C. The Derivative of a Regular Type Is Its Type of One-Hole Contexts, Unpublished, 2001, http://strictlypositive.org/diff.pdf

[11] Jones, S. P and Vytiniotis. D and Weirich S, and Washburn G. Simple unification-based type inference for GADTs, in ICFP, 2006

[12] Chakravarty, M. M. T. and Keller, G. and Jones, S. P. and Marlow, S. Associated types with class, In POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, http://www.cse.unsw.edu.au/ chak/papers/assoc.ps.gz, 2005

Typed Transformations of Typed Grammars: The Left Corner Transform

Arthur Baars [1]

Instituto Tecnológico de Informática
Universidad Politécnica de Valencia, Valencia, Spain

S. Doaitse Swierstra [2]

Department of Computer Science
Utrecht University, Utrecht, The Netherlands

Marcos Viera [3]

Instituto de Computación
Universidad de la República, Montevideo, Uruguay

Abstract

One of the questions which comes up when using embedded domain specific languages is to what extent we can analyze and transform embedded programs, as normally done in more conventional compilers. Special problems arise when the host language is strongly typed, and this host type system is used to type the embedded language. In this paper we describe how we can use a library, which was designed for constructing transformations of typed abstract syntax, in the removal of left recursion from a typed grammar description. The algorithm we describe is the Left-Corner Transform, which is small enough to be fully explained, involved enough to be interesting, and complete enough to serve as a tutorial on how to proceed in similar cases. The described transformation has been successfully used in constructing a compositional and efficient alternative to the standard Haskell *read* function.

Key words: GADT, Left-Corner Transform, Meta Programming, Type Systems, Typed Abstract Syntax, Typed Transformations

[1] Email: abaars@iti.upv.es
[2] Email: doaitse@cs.uu.nl
[3] Email: mviera@fing.edu.uy

1 Introduction

In Haskell one can describe how values of a specific data type are to be serialised (i.e. written) and deserialised (i.e. read or parsed). Since data types can be passed as parameter to data type constructors, and definitions can be spread over several modules, the question arises how to dynamically combine separately generated pieces of "reading code" into a function *read* for a composite data type. The standard solution in Haskell, based on a straightforward combination of top-down parsers, has turned out to exhibit exponential reading times. Furthermore, in order to avoid the dynamic construction of left recursive top-down parsers at run-time the input is required to contain many more parentheses than one would expect.

In a recent paper [12] we have presented a solution to this problem; instead of generating code which reads a value of some data type a, the compiler constructs a value of type $Grammar\ a$ which represents the piece of grammar that describes external representations of values of that data type. The striking feature of this grammar type is that it reflects the type of values represented. This is necessary, since from such a value eventually a *read* function of type $String \to a$ has to be constructed by Haskell library code.

The solution builds upon three, more or less independent, layers (from top to bottom):

(i) A template Haskell library which generates the values of type $Grammar\ a$ and library code which combines such values at run-time to form a complete grammar. Out of this combined value the desired read function for the composed data type is constructed, again by library Haskell code. This whole process is described in the aforementioned paper [12].

(ii) This code calls a library function which removes potential left-recursion from the composed grammar. For this we use the Left-Corner Transform (LCT) [7]. This code, which produces a function of type $Grammar\ a \to Grammar\ a$, is a fine example of how to express transformations of typed abstract syntax containing references; in the $Grammar\ a$ case these stem from occurrences of non-terminal symbols in the right hand sides of the productions.

(iii) The LCT and the left-factoring code make use of an intricate Haskell library, which exploits every corner of the Haskell type system and its extensions, such as Generalised Algebraic Data Types, existential and polymorphic types, and lazy evaluation at the type level. The design alternatives and the final design of the library, as it has been made available to the Haskell world, deserved a paper of its own [1].

In this paper we focus on the middle of the above three layers; we start out by presenting an elegant formulation of the LCT in combination with an untyped Haskell implementation, next we introduce the API as implemented by the bottom layer, and we finish by reformulating the untyped version into

a typed one using this API.

The LCT [6] is more involved than the direct left recursion removal given in [2], but is also more efficient ($O(n^2)$, where n is the number of terminals and non-terminals in the grammar). Here we will start from an improved version formulated by Robert C. Moore [7], which we present in a more intuitive form. Both his tests, using several large grammars for natural language processing, and our tests [12], using several very large data type descriptions, show that the algorithm performs very well in practice.

What makes this transformation interesting from the typed abstract syntax point of view is that a grammar consists of a collection of grammar rules (one for each non-terminal) *containing references to other definitions*; we are thus not transforming a tree but a complete binding structure. During this transformation we introduce many new definitions. In the right hand side of these definitions we again use references to such newly introduced symbols. In our setting a transformation must be type preserving and we thus have to ensure that the types of the environment and the references remain consistent, *while being modified*. Previous work on typeful program transformations [3,8,2] cannot handle such introductions of new definitions and binders.

We present the algorithm in terms of Haskell code, and thus require Haskell knowledge from the reader. Please keep in mind however that Haskell currently is one of the few general purpose languages in which the problem we describe can be solved at all.

2 Left-Corner Transform

In this section we introduce the LCT [6] as a set of construction rules and subsequently give an untyped implementation in Haskell98. Note that, despite being called a transformation, the process is actually constructing a new grammar while inspecting the input grammar. We assume that only the start symbol may derive ϵ.

We say that a symbol X is a *direct left-corner* of a non-terminal A, if there exists a production for A which has the symbol X as its left-most symbol in the right-hand side of that production. We define the *left-corner* relation as the transitive closure of the direct left-corner relation. Note that a non-terminal being left-recursive is equivalent to being a left-corner of itself.

The LCT is defined as the application of three surprisingly simple rules. We use lower-case letters to denote terminal symbols, low-order upper-case letters (A, B, etc.) to denote non-terminals from the grammar and high-order upper-case letters (X, Y, Z) to denote symbols that can either be terminals or non-terminals. Greek symbols denote sequences of terminals and non-terminals.

For a non-terminal A of the original grammar the algorithm constructs new productions for A, and a set of new definitions for non-terminals of the form A_X. A new non-terminal A_X represents that part of A which is still

to be recognised after having seen an X. The following rules are applied for each non-terminal until no further results are obtained:

Rule 1 For each production $A \to X\ \beta$ of the original grammar add $A\_X \to \beta$ to the transformed grammar, and add X to the left-corners of A.

Rule 2 For each newly found left-corner X of A:

a If X is a terminal symbol add $A \to X\ A\_X$ to the new grammar.

b If X is a non-terminal then for each original production $X \to X'\ \beta$ add the production $A\_X' \to \beta\ A\_X$ to the new grammar and add X' to the left-corners of A.

As an example consider the grammar:

$A \to a\ A \mid B$
$B \to A\ b \mid c$

Applying rule 1 for the productions of A results in two new productions and two newly encountered left-corners:

$A\_a \to A$
$A\_B \to \epsilon$ $\quad leftcorners = [a, B]$

rule 2a with X bound to the left-corner $a \Rightarrow$

$A \to a\ A\_a$ $\quad leftcorners = [\sout{a}, B]$

rule 2b with X bound to the left-corner $B \Rightarrow$

$A\_A \to b\ A\_B$
$A\_c \to A\_B$ $\quad leftcorners = [\sout{a}, \sout{B}, A, c]$

rule 2b with X bound to the left-corner $A \Rightarrow$

$A\_a \to A\ A\_A$
$A\_B \to A\_A$ $\quad leftcorners = [\sout{a}, \sout{B}, \sout{A}, c]$

rule 2a with X bound to the left-corner $c \Rightarrow$

$A \to c\ A\_c$ $\quad leftcorners = [\sout{a}, \sout{B}, \sout{A}, \sout{c}]$

Since now all left-corners of A have been processed we are done with A. For the non-terminal B the process yields the following new productions:

$B\_A \to b$ -- rule 1
$B\_c \to \epsilon$ -- rule 1
$B\_a \to A\ B\_A$ -- rule 2b, A
$B\_B \to B\_A$ -- rule 2b, A
$B \to c\ B\_c$ -- rule 2a, c
$B \to a\ B\_a$ -- rule 2a, a
$B\_A \to b\ B\_B$ -- rule 2b, B
$B\_c \to B\_B$ -- rule 2b, B

Note that by construction this new grammar is not left-recursive.

2.1 The Untyped Left-Corner Transform

Before presenting our typed LCT, we present an untyped implementation. Grammars are represented by the types:

```
type Grammar = Map NT [Prod]
type NT      = String
type Prod    = [Symbol]
type Symbol  = String
isNonterminal = isUpper . head
isTerminal    = isLower . head
```

Thus a *Grammar* is a mapping which associates each non-terminal name with its set of productions. Each production (*Prod*) consists of a sequence of symbols (*Symbol*). So our example grammar can be encoded as:

```
grammar = Map.fromList [("A", [["a", "A"], ["B"]])
                       ,("B", [["A", "b"], ["c"]])]
```

In the transformation process we use the *Control.Monad.State*-monad to store the thus far constructed new grammar. For each non-terminal we traverse the transitive left-corner relation as induced by the productions in depth-first order, while caching the set of thus far encountered left-corner symbols in a list:

```
type LeftCorner = Symbol
type Step_State = (Grammar, [LeftCorner])
type Trafo a = State Step_State a
```

The function *leftcorner* takes a grammar and returns a transformed grammar by running the transformation *rules1*, which yields a value of the monadic type *Trafo*. The state is initialized with an empty grammar and an empty list of encountered left-corner symbols. The final state contains the newly constructed grammar:

```
leftcorner :: Grammar → Grammar
leftcorner g = fst . snd . runState (rules1 g g) $ (Map.empty, [])
```

For each ($mapM\_$) non-terminal (A) the function *rules1* visits each ($mapM$) of its productions; each visit results in new productions using *rule2a* and *rule2b*. They are added to the transformed grammar by the function *insert*. The productions resulting from *rule2a* are returned (ps), and together ($concat$) from the new productions for the original non-terminal A. The left-corners cache is reset when starting with the next non-terminal:

```
rules1 :: Grammar → Grammar → Trafo ()
rules1 gram nts = mapM_ nt (Map.toList nts)
  where nt (a, prods) =
              do ps ← mapM (rule1 gram a) prods
                 modify  (λ(g, _) → (Map.insert a (concat ps) g, []))
```

For each of the rules given we define a function: *rule2b* generates new productions for non-terminals of the original grammar, and *rule1* and *rule2b* generate productions for non-terminals of the form A_X:

```
rule1 :: Grammar → NT → Prod → Trafo [Prod]
rule1 grammar a (x : beta) = insert grammar a x beta

rule2a :: NT → Symbol → Prod
rule2a a_b b = [b, a_b]
```

```
rule2b :: Grammar → NT → NT → Prod → Trafo [Prod]
rule2b grammar a a_b (y : beta) = insert grammar a y (beta ++ [a_b])
```

The function *insert* adds a new production for a non-terminal A_X to the grammar: if we have met A_X before, the already existing entry is extended and otherwise a new entry for A_X is added. In the latter case we apply *rule2* in order to find further left-corner symbols:

```
insert :: Grammar → NT → Symbol → Prod → Trafo [Prod]
insert grammar a x p =
  do let a_x = a ++ "_" ++ x
     (gram, lcs) ← get
     if x ∈ lcs then do put (Map.adjust (p:) a_x gram, lcs)
                        return []
                else  do put (Map.insert a_x [p] gram, x : lcs)
                        rule2 grammar a x
```

In *rule2* new productions resulting from applications of *rule2b* are directly inserted into the transformed grammar, whereas the productions resulting from *rule2a* are collected and returned as the result of the *Trafo*-monad. When the newly found left-corner symbol is a terminal *rule2a* is applied, and the resulting new production rule is simply returned. If it is a non-terminal, its corresponding productions are located in the original grammar and *rule2b* is applied to each of them:

```
rule2 :: Grammar → NT → Symbol → Trafo [Prod]
rule2 grammar a b
   | isTerminal b = return [rule2a a_b b]
   | otherwise    = do let Just prods = Map.lookup b grammar
                       rs ← mapM (rule2b grammar a a_b) prods
                       return (concat rs)
  where a_b = a ++ "_" ++ b
```

Note that the functions *rule2* and *insert* are mutually recursive. They apply the rules 2a and 2b until no new left-corner symbols are found. The structure of the typed implementation we present in section 4 closely resembles the untyped solution above.

3 Typed Transformations

The typed version of the LC transform is implemented by using a library (TTTAS [4]) we described in a companion paper [1] to perform typed transformations of typed abstract syntax (in our case typed grammars). In the following subsections we introduce the basic constructs for representing typed abstract syntax and the library interface for manipulating it.

[4] `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/TTTAS`.

3.1 Typed References and Environments

Pasalic and Linger [8] introduced an encoding *Ref* of typed references pointing into an environment containing values of different type. A *Ref* is actually an index labeled with both the type of the referenced value and the type of the environment (a nested Cartesian product, growing to the right) the value lives in:

```
data Ref a env where
  Zero ::                  Ref a (env', a)
  Suc  :: Ref a env' → Ref a (env', b)
```

The type *Ref* is a generalized algebraic data type [10]. The constructor *Zero* expresses that the first element of the environment has to be of type a. The constructor *Suc* does not care about the type of the first element in the environment (it is polymorphic in b), and remembers a position in the rest of the environment.

We extend this idea such that environments do not contain values of mixed type but *terms* (expressions) describing such values instead; these terms take an extra type parameter describing the environment into which references to other terms occurring in the term may point. In this way we can describe typed terms containing typed references to other terms. As a consequence, an *Env* may be used to represent an environment, consisting of a collection of possibly mutually recursive definitions. The environment stores a heterogeneous list of terms of type *t a use*, which are the right-hand expressions of the definitions. References to elements are represented by indices in the list.

```
data Env term use def where
  Empty :: Env t use ()
  Ext   :: Env t use def' → t a use → Env t use (def', a)
```

The type parameter *def* contains all the type labels a of the terms of type *t a use* occurring in the environment. When a term is added to the environment using *Ext*, its type label is included as the first component of *def*. The type *use* describes the types that may be referenced from within terms of type *t a use* using *Ref a use* values. When the types *def* and *use* coincide the type system ensures that the references in the terms do not point to values outside the environment.

The function *lookupEnv* takes a reference and an environment into which the reference points. The occurrence of the two *env*'s in the type of *lookupEnv* guarantees that the lookup will succeed, and that the value found is indeed labeled with the type with which the *Ref* argument was labeled, which is encoded by the two occurrences of a:

```
lookupEnv :: Ref a env → Env t s env → t a s
lookupEnv Zero    (Ext _ t)  = t
lookupEnv (Suc r) (Ext ts _) = lookupEnv r ts
```

3.2 Transformation Library

The library is based on the type *Trafo*, which represents typed transformation steps. Each transformation step (possibly) extends an implicitly maintained environment *Env*.

```
data Trafo m t s a b = Trafo (∀env1 . m env1 → TrafoE m t s env1 a b)
```

The argument m stands for the type of the observable state of the transformation. A *Trafo* takes such a state value, which depends on the environment constructed thus far, as input and yields a new state corresponding to the (possibly extended) environment. The type t is the type of the terms stored in the environment. The type variable s represents the type of the final result, which is passed as the *use* argument in the embedded references. We compose transformations in an arrow style. The arguments a and b are the *Arrow*'s input and output, respectively. The *Arrow* library [5] contains a set of functions for constructing and combining values that are instance of the *Arrow* class. Furthermore there is a convenient notation [9] for programming with *Arrow*s. This notation is inspired by the **do**-notation for *Monad*s. The class *ArrowLoop* is instantiated to be able to construct feedback loops. The TTTAS library includes a combinator, analogous to the *sequence* combinator for *Monad*s, which combines a sequence of transformations into one single large transformation:

```
sequenceA :: [Trafo m t s a b] → Trafo m t s a [b]
```

Each individual transformation maps the input a onto a value b. The combined results b resulting from applying the individual transformations in sequence, are returned as a list $[b]$.

The constructor *Trafo* contains a function which maps a state in the current environment to the actual transformation, represented by the type *TrafoE*. Because the internal details of the type *TrafoE* are of no relevance here, we do not give its definition; we only present its constructors:

```
extEnv      :: m (e, a)          → TrafoE m t s e (t a s) (Ref a s)
castSRef    :: m e → Ref a e → TrafoE m t s e i       (Ref a s)
updateSRef :: m e → Ref a e → (i → t a s → t a s)
                                 → TrafoE m t s e i       (Ref a s)
```

The function *extEnv* builds a *TrafoE* which takes a typed term (of type $t\ a\ s$) as input, adds it to the environment and yields a reference pointing to this value in the final environment (s). The argument of *extEnv* is a state that depends on the extended environment (e, a). Thus, for example, a transformation that extends the environment without keeping any internal state can be implemented:

```
data Unit env = Unit

newSRef :: Trafo Unit t s (t a s) (Ref a s)
newSRef = Trafo (λ_ → extEnv Unit)
```

The function *castSRef* builds a *TrafoE* that returns the reference passed as parameter (in the current environment e) casted to the final environment. The

function *updateSRef* builds a *TrafoE* that updates the value pointed by the passed reference. Note that the update function (of type $i \to t\ a\ s \to t\ a\ s$) can use the input of the *Arrow*. The type ($TrafoE\ m\ t\ s\ e\ a$) is an instance of the class *Functor*, so the function

$$fmap :: (b \to c) \to TrafoE\ m\ t\ s\ e\ a\ b \to TrafoE\ m\ t\ s\ e\ a\ c$$

lifts a function with type $(b \to c)$ and applies it to the output of the *Arrow*.

When we run a transformation we start with an empty environment and an initial value. Since this argument type is labeled with the final environment, which we do not know yet, is has to be a polymorphic value.

$$runTrafo :: (\forall s\ .\ Trafo\ m\ t\ s\ a\ (b\ s)) \to m\ () \to a \to Result\ m\ t\ b$$

The *Result* contains the final state ($m\ s$), the output value ($b\ s$) and the final environment ($Env\ t\ s\ s$). Since in general we do not know how many new definitions and of which types are introduced by the transformation the result is existential in the final environment s. Despite this existentially, we can enforce the environment to be closed:

$$\textbf{data}\ Result\ m\ t\ b = \forall s\ .\ Result\ (m\ s)\ (b\ s)\ (Env\ t\ s\ s)$$

4 The Typed Left-Corner Transform

For a typed version of the LCT we need a typed representation of grammars. A grammar consists of a start symbol, represented as a reference labeled with the type that serves as the witness value of a successful parse, and an *Env*, containing for each non-terminal its list of productions. The actual type *env*, describing the types associated with the non-terminals, is hidden using existential quantification:

$$\begin{aligned}&\textbf{data}\ Grammar\ a = \forall env\ .\ Grammar\ (Ref\ a\ env)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (Env\ Productions\ env\ env)\\ &\textbf{newtype}\ Productions\ a\ env = PS\{\ unPS :: [\,Prod\ a\ env\,]\ \}\end{aligned}$$

Since in our LCT we want to have easy access to the first symbol of a production we have chosen a representation which facilitates this. Hence the types of the elements in a sequential composition have been chosen a bit different from the usual one [11], such that *Seq* can be chosen to be right associative. The types have been chosen in such a way that if we close the right hand side sequence of symbols with an $End\ f$ element, then this f is a function that accepts the results of the earlier elements (parsing results of the right hand side) as arguments, and builds the parsing result for the left-hand side non-terminal. In our case a production is a sequence of symbols, and a symbol is either a terminal with a *String* as its witness or a non-terminal (reference):

$$\begin{aligned}&\textbf{data}\ Symbol :: * \to * \to *\ \textbf{where}\\ &\quad Nont :: Ref\ a\ env \to Symbol\ a\ \quad env\\ &\quad Term :: String \to \quad Symbol\ String\ env\\ &\textbf{data}\ Prod :: * \to * \to *\ \textbf{where}\\ &\quad Seq\ :: Symbol\ b\ env \to Prod\ (b \to a)\ env \to Prod\ a\ env\end{aligned}$$

```
End :: a                  → Prod  a        env
```

In order to make our grammars resemble normal grammars we introduce some extra operators:

```
infixr 5 `cons`, .*.

cons prods g = Ext g (PS prods)
(.*.) = Seq
```

We now have the machinery at hand to encode our example grammar:

```
_A = Nont      Zero
_B = Nont (Suc Zero)
_a = Term "a"
_b = Term "b"
_c = Term "c"
```

Assume we want the witness type for non-terminal A to be a *String* and for non-terminal B an *Int*:

```
grammar :: Grammar String
grammar = Grammar Zero productions

type Types_nts = (((), Int), String)

productions :: Env Productions Types_nts Types_nts
productions = [_a .*. _A .*. End (++)
              ,_B        .*. End show     ] `cons`
              [_A .*. _b .*. End (λy x → length x + length y)
              ,_c        .*. End (const 1)] `cons` Empty
```

Before delving into the LCT itself we introduce some grammar related functions we will need:

```
append :: (a → b → c) → Prod a env → Symbol b env → Prod c env
matchSym :: Symbol a env → Symbol b env → Maybe (Equal a b)
mapProd :: T env1 env2 → Prod a env1 → Prod a env2
```

The function *append* is used in the LCT to build productions of the form $\beta\ X\_A$. Basically it corresponds to the *snoc* operation on lists; we only have to make sure that all the types match. The function *matchSym* compares two symbols and, if they are equal, returns a witness (*Equal*) of the proof that the types a and b are equal. The function *mapProd* systematically changes all the references to non-terminals occurring in a production. It takes a *Ref*-transformer (*T env1 env2*) to transform references in the environment *env1* to references in the environment *env2*.

```
newtype T env1 env2 = T{ unT :: ∀x . Ref x env1 → Ref x env2 }
```

4.1 The Typed Transformation

The LCT is applied in turn to each non-terminal (A) of the original grammar. The algorithm performs a depth first search for left-corner symbols. For each left-corner X a new non-terminal A_X is introduced. Additionally a new definition for A itself is added to the transformed grammar.

In the untyped implementation we simply used strings to represent non-terminals. In the typed solution non-terminals are, however, represented as typed references. The first time a production for a non-terminal A_X is generated, we must create a new entry for this non-terminal and remember its position. When the next production for such an A_X is generated we must add it to the already generated productions for this A_X: hence we maintain a finite map from encountered left-corner symbols (X) to references corresponding to the non-terminals (A_X). This finite map again caches the already encountered left-corner symbols:

```
newtype MapA_X env a env2
  = MapA_X (∀x . Symbol x env → Maybe (Ref (x → a) env2))
```

The type variable *env* comes from the original grammar, and *env2* is the type of the new grammar constructed thus far. The type variable a is the type of the current non-terminal. A left-corner symbol labelled with type x is mapped to a reference to the definitions of the non-terminal A_X in the new grammar, provided it was inserted earlier. The type associated with a non-terminal of the form A_X is $(x \to a)$, i.e. a function that returns the semantics of A, when it is passed the semantics of the symbol X. The empty mapping is defined as:

```
emptyMap :: MapA_X env a env2
emptyMap = MapA_X (const Nothing)
```

We introduce the type-synonym *LCTrafo*, which is the type of the transformation step of the LCT. The type of our terms is *Productions*, and the internal state is a table of type *MapA_X*, containing the encountered left-corner symbols.

```
type LCTrafo env a = Trafo (MapA_X env a) Productions
```

Next we define the function *newNontR* which is a special version of the function *newSRef*, using *MapA_X* as internal state instead of *Unit*. It takes a left-corner symbol X as argument and yields a *LCTrafo* that introduces a new non-terminal A_X. The input of the *LCTrafo* is the first production (*Productions*) for A_X, and the output is the reference to this newly added non-terminal:

```
newNontR :: ∀x env s a . Symbol x env
            → LCTrafo env a s (Productions (x → a) s) (Ref (x → a) s)
newNontR x = Trafo $ λm → extEnv (extendMap x m)
```

The symbol X is added to the map of encountered left-corners of A by the function *extendMap*, which records the fact that the newly founded left-corner is the first element of the environment (*Zero*) and the previously added ones have to be shifted one place (*Suc*).

```
extendMap :: Symbol x env → MapA_X env a env'
             → MapA_X env a (env', x → a)
extendMap x (MapA_X m) = MapA_X (λs → case matchSym s x of
                                        Just Eq → Just Zero
                                        Nothing → fmap Suc (m s))
```

The index at which the new definition for A is stored is usually different from the index of A in the original grammar. This is a problem as we need to copy parts (the βs in the rules) of the original grammar into the new grammar. The non-terminal references in these parts must be adjusted to the new indexes. To achieve this we first collect all the new references for the non-terminals of the original grammar into a finite map, and then use this map to compute a *Ref*-transformer that is subsequently passed around and used to convert references from the original grammar to corresponding references in the new grammar. The type of this finite map is:

```
newtype Mapping o n = Mapping (Env Ref n o)
```

The mapping is represented as an *Env*, and contains for each non-terminal of the old grammar, the corresponding reference in the new grammar. The mapping can easily be converted into a *Ref*-transformer:

```
map2trans :: Mapping env s → T env s
map2trans (Mapping env) = T (λr → (lookupEnv r env))
```

Now all that is left to do is to glue all the pieces defined above together. Each of the following functions corresponds to the untyped version with the same name. We start with the function *insert*:

```
insert :: ∀env s a x . Env Productions env env → Symbol x env
       → LCTrafo env a s (T env s, Prod (x → a) s) (Productions a s)
insert old_gram x =
  Trafo  (λ(MapA_X m) →
            case m x of
              Just r    → extendA_X (MapA_X m) r
              Nothing → insNewA_X (MapA_X m))
  where
    Trafo insNewA_X = proc (tenv_s, p) → do
      r ← newNontR x ≺ PS [p]
      rule2 old_gram x ≺ (tenv_s, r)
```

This function takes the original grammar and a left-corner symbol x as input. It yields a transformation that takes as input a *Ref*-transformer from the original to the new (transformed) grammar and a production for the non-terminal A_X, and stores this production in the transformed grammar. If the symbol x is new ($m\ x$ returns *Nothing*), the production is stored at a new index (using *newNontR*) and the function *rule2* is applied, to continue the depth-first search for left-corners. If we already know that x is a left-corner of a then we obtain an index r to the previously added to the non-terminal A_X, and add the new production at this position. The function *extendA_X* returns the *TrafoE* that performs this update into the environment:

```
extendA_X ::  m env1 → Ref (x → a) env1
          → TrafoE m Productions s env1 (T env s, Prod (x → a) s)
                                        (Productions a s)
extendA_X m r = fmap (const $ PS []) $ updateSRef m r addProd
          where addProd (_, p) (PS ps) = PS (p : ps)
```

If in the function *rule2* the left-corner is a terminal symbol then *rule2a* is applied, and the new production rule is returned as *Arrow*-output. In case the left-corner is a non-terminal the corresponding productions are looked up in the original grammar, and *rule2b* is applied to all of them, thus extending the grammar under construction:

```
rule2 :: Env Productions env env → Symbol x env
      → LCTrafo env a s (T env s, Ref (x → a) s) (Productions a s)
rule2 _ (Term a) = proc (_, a_x) → returnA ≺ PS [rule2a a a_x]
rule2 old_gram (Nont b) = case lookupEnv b old_gram of
  PS ps → proc (tenv_s, a_x) → do
            pss ← sequenceA (map (rule2b old_gram) ps) ≺ (tenv_s, a_x)
            returnA ≺ PS (concatMap unPS pss)
```

We now define the functions *rule2a*, and *rule2b* that implement the corresponding rules of the LCT. Firstly, *rule2a*, which does not introduce a new non-terminal, but simply provides new productions for the non-terminal (A) under consideration. The implementation of rule 2a is as follows:

```
rule2a :: String → Ref (String → a) s → Prod a s
rule2a a refA_a = Term a . ∗ . Nont refA_a . ∗ . End ($)
```

The function *rule2b* takes the original grammar and a production from the original grammar as arguments, and yields a transformation that takes as input a *Ref*-transformer and a reference for the non-terminal A_B, and constructs a new production which is subsequently inserted. Note that the *Ref*-transformer *tenv_s* is applied to the non-terminal references in *beta* to map them on the corresponding references in the new grammar.

```
rule2b :: Env Productions env env → Prod b env
       → LCTrafo env a s (T env s, Ref (b → a) s) (Productions a s)
rule2b old_gram (Seq x beta)
  = proc (tenv_s, a_b) →
    insert old_gram x ≺ (tenv_s
                        , append (flip (.)) (mapProd tenv_s beta) (Nont a_b)
                        )
```

The function *rule1* is almost identical to *rule2b*; the only difference is that it deals with direct left-corners and hence does not involve a "parent" non-terminal A_B.

```
rule1 :: Env Productions env env → Prod a env
      → LCTrafo env a s (T env s) (Productions a s)
rule1 old_gram (Seq x beta)
  = proc tenv_s →
    insert old_gram x ≺ (tenv_s, mapProd tenv_s beta)
```

The function *rules1* is defined by induction over the original grammar (i.e. it iterates over the non-terminals) with the second parameter as the induction parameter. It is polymorphically recursive: the type variable env' changes during induction, starting with the type of the original grammar (i.e. *env*) and ending with the type of the empty grammar (). The first argument is a

copy of the original grammar which is needed for looking up the productions of the original non-terminals:

```
rules1 :: Env Productions env env → Env Productions env env'
       → Trafo Unit Productions s (T env s) (Mapping env' s)
rules1 _ Empty = proc _ → returnA ≺ Mapping Empty
rules1 old_gram (Ext ps (PS prods)) = proc tenv_s → do
          p ← initMap nt ≺ tenv_s
          r ← newSRef ≺ p
          Mapping e ← rules1 old_gram ps ≺ tenv_s
          returnA ≺ Mapping (Ext e r)
  where
    nt = proc tenv_s → do
               pss ← sequenceA (map (rule1 old_gram) prods) ≺ tenv_s
               returnA ≺ PS (concatMap unPS pss)
```

The result of *rules1* is the complete transformation represented as a value of type *Trafo*. At the top-level the transformation does not use any state, hence the type *Unit*. When dealing with one non-terminal (*nt*), *rule1* is applied for each of its productions and the new productions are collected to be inserted in the new grammar. The function *initMap* initialises the state information of the transformation *nt* with an empty table of encountered left-corners.

```
initMap :: LCTrafo env a s c d → Trafo Unit Productions s c d
initMap (Trafo st) = Trafo (λ_ → case st emptyMap of
                                   TrafoE _ f → TrafoE Unit f)
```

As input the transformation returned by *rules1* needs a *Ref*-transformer to remap non-terminals of the old grammar to the new grammar. During the transformation *rules1* inserts the new definitions for non-terminals of the original grammar, and remembers the new locations for these non-terminals in a *Mapping*. This *Mapping* can be converted into the required *Ref*-transformer, which must be fed-back as the *Arrow*-input. This feed-back loop is made in the function *leftcorner* using **mdo**-notation:

```
leftcorner :: ∀a . Grammar a → Grammar a
leftcorner (Grammar start productions)
    = case runTrafo lctrafo Unit ⊥ of
      Result _ (T tt) gram → Grammar (tt start) gram
  where
    lctrafo = proc _ → mdo
      let tenv_s = map2trans menv_s
      menv_s ← (rules1 productions productions) ≺ tenv_s
      returnA ≺ tenv_s
```

The resulting transformation is run using $\bot$ as input; this is perfectly safe as it does not use the input at all: the result is a new start symbol and the transformed production rules, which are combined to form the new grammar.

5 Conclusions

We have shown how complicated transformations can be done at run-time, while having been partially verified statically by the type system. Doing so we have used a wide variety of type system concepts, like GADTs and existential and polymorphic types, which cannot be found together in other general purpose languages than Haskell. This allows us to use techniques which are typical of dependently typed systems while maintaining a complete separation between types and values. Besides this we make use of lazy evaluation in order to get computed information to the right places to be used.

Implementing transformations like the left-corner transform implies the introduction of new references to a collection of possibly mutually recursive definitions. Previous work on typeful transformations of embedded DSLs represented as typed abstract syntax [3,2,4] does not deal with such complexity. Thus, as far as we know, this is the first description of run-time typed transformations which modify references into an abstract syntax represented as a graph instead of a tree.

We have shown how the untyped version of a transformation can be transformed into a typed version; after studying this example the implementation of similar transformations, using the TTTAS library, should be relatively straightforward. Despite the fact that this transformation is rather systematic, it remains a subject of future research to see how such transformations can be done automatically.

References

[1] Arthur Baars, S. Doaitse Swierstra, and Marcos Viera. Typed transformations of typed abstract syntax. In *TLDI '09: fourth ACM SIGPLAN Workshop on Types in Language Design and Implementation*, New York, NY, USA, 2009. ACM.

[2] Arthur I. Baars and S. Doaitse Swierstra. Type-safe, self-inspecting code. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 69–79, New York, NY, USA, 2004. ACM Press.

[3] Chiyan Chen and Hongwei Xi. Implementing typeful program transformations. In *PEPM'03*, 2003.

[4] Louis-Julien Guillemette and Stefan Monnier. Type-safe code transformations in haskell. *Electron. Notes Theor. Comput. Sci.*, 174(7):23–39, 2007.

[5] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

[6] M. Johnson. Finite-state approximation of constraint-based grammars using left-corner grammar transforms. In *COLING-ACL 98, Montreal, Quebec, Canada*, pages 619–623. Association for Computational Linguistics, 1998.

[7] Robert C. Moore. Removing left recursion from context-free grammars. In *Proceedings of the first conference on North American chapter of the Association for Computational Linguistics*, pages 249–255, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[8] Emir Pasalic and Nathan Linger. Meta-programming with typed object-language representations. In *Generative Programming and Component Engineering (GPCE'04)*, volume LNCS 3286, pages 136 – 167, October 2004.

[9] Ross Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, September 2001.

[10] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. *SIGPLAN Not.*, 41(9):50–61, 2006.

[11] Doaitse Swierstra and Luc Duponcheel. Deterministic, error correcting combinator parsers. In *Advanced Functional Programming, Second International Spring School*, volume 1129 of *LNCS*, pages 184–207. Springer-Verlag, 1996.

[12] Marcos Viera, S. Doaitse Swierstra, and Eelco Lempsink. Haskell, do you read me?: constructing and composing efficient top-down parsers at runtime. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 63–74, New York, NY, USA, 2008. ACM.

Safely Speaking in Tongues

Statically Checking Domain Specific Languages in Haskell

Matthew Sackman [1]

Imperial College,
London, England

Susan Eisenbach [2]

Imperial College,
London, England

Abstract

Haskell makes it very easy to build and use Domain Specific Languages (DSLs). However, it is frequently the case that a DSL has invariants that can not be easily enforced statically, resulting in runtime checks. This is a great pity given Haskell's rich and powerful type system and leads to all the usual problems of dynamic checking.

We believe that Domain Specific Languages are becoming more popular: the internet itself is a good example of many DSLs (HTML, CSS, JavaScript, Flash, etc), and more seem to be being added every day; most graphics cards already accept programs written in the DSL OpenGL Shading Language (GLSL); and the predicted growth of heterogeneous CPUs (for example IBM's Cell CPU) will demand many different DSLs for the various programming models and instruction sets that become available.

We present a technique that allows many invariants of different DSLs to be lifted into the Haskell type system. This removes the need for runtime checks of the DSL and prevents programs that violate the invariants of the DSL from ever being compiled or executed. As a result we avoid the pitfalls of dynamic checking and return the user of the DSL to the safety and tranquillity of the strongly statically typed Haskell world.

Key words: Domain Specific Languages, Type Systems, Static Verification, Language Invariants, Haskell

[1] Email: ms@doc.ic.ac.uk
[2] Email: sue@doc.ic.ac.uk

1 Introduction

Algebraic Data Types (ADTs) provide a very easy way to design and build a Domain Specific Language (DSL) with some degree of type safety. For example, a simple expression language can be created very easily:

```
data Expr = Bool Bool       | Int Int
          | Plus Expr Expr | Times Expr Expr | Eq Expr Expr
```

However, this does not prevent a *Bool* being compared for equality (*Eq*) with an *Int*, which we might well want to reject rather than just return false. Even worse, we can attempt to multiply an *Int* by a *Bool*, which we definitely want to reject. Here the problem is that we have two distinct types, integers and booleans, and we must limit the constructors to only accepting expressions of the correct type. Thus we add a type parameter to track the type of the current expression and we switch to using Generalised Algebraic Data Types (GADTs) [1]. GADTs allow type signatures of constructors to be specified explicitly, and type parameters need not appear in constructor arguments. Using a GADT rather than several plain ADTs offers the simplest solution as we don't need to build a hierarchy of data types:

```
data Expr t where
  Bool  :: Bool → Expr Bool
  Int   :: Int → Expr Int
  Plus  :: Expr Int → Expr Int → Expr Int
  Times :: Expr Int → Expr Int → Expr Int
  Eq    :: Expr t → Expr t → Expr Bool
```

Eq is particularly satisfying here as it allows any two expressions of the same type to be compared for equality, thus there is no need for separate *Eq* constructors for boolean and integer expressions.

If the DSL is to contain statements rather than expressions, and has state, then things can become rather more tricky. Making further use of GADTs to model variables and variable creation, and combining these GADTs with the *State* monad certainly allows you to statically enforce some invariants. But not all desired invariants can be enforced with these techniques. For example, consider a DSL for the assembly language with CPU-registers. The value within a register has a type, and you may wish to ensure that you can only use type-safe operators with that register. But the register could load another value with another type, thus the type of the register has to be able to vary. The GADT approach cannot automatically consider that the register can change type and so would require explicit constructs in the DSL to permit this. Another example is the security analysis (the Bell / LaPadula system) that we present in section 5: this cannot be implemented through GADTs alone.

It is very common then for GADTs and related techniques to be used to enforce as much as possible, but then for certain checks to be performed at runtime. This often provides a false sense of security to the user of the

DSL as they may not realise that their DSL-program may be rejected at runtime, and it also complicates matters significantly for the library writer and DSL-designer as they must ensure that they check the DSL-program at every necessary point in the provided API. Forgetting to check the DSL in just one public function may be enough to expose privileged information or to at least bring down a system. Further, if the same DSL-program is used in multiple places, the checking would have to be done multiple times. This could become very expensive if the DSL-program is large and the checking complex. It should not be difficult to see the benefits of being able to statically check the DSL-program.

- We present a technique that allows such complex invariants to be statically checked by lifting the invariants into the Haskell type system.
- We demonstrate our technique with a running example using an extended form of the While language as our DSL (section 2) which permits both boolean and numeric variables and for variables to change their type. After explaining our framework (section 3) we use it to implement type inference (section 4) and then extend the While language with a security model (Bell / LaPadula) which gives secrecy levels to variables and enforces restrictions on information flow (section 5).

DSLs are not a new technique and have been successfully used for a variety of applications. SQL is a very widely used DSL for accessing databases and has been subjected to many techniques to statically enforce its invariants. We discuss these and other related work in section 6, before concluding in section 7.

2 The While Language

We demonstrate our technique by enforcing invariants on a simple While language. The While language is a very basic but general purpose imperative language, that is statement based using variable assignment (`:=`), has typical arithmetic, boolean and logical functions, and has `while` and `if` control flow constructs. The `if` must have both branches specified and there is a no-op instruction, `skip` which is typically used in a branch of an `if` to effectively eliminate that branch. Using a general purpose language (albeit it a very simple one) as our DSL demonstrates how general our technique is, and offers evidence that our techniques can be adapted and used in far more specific DSLs.

The While language as defined in [2] only permitted variables to be assigned numeric expressions and boolean expressions only appeared in the conditions of `while` and `if` statements. We extend the While language so that boolean expressions can be assigned to variables, and that variables can change their type, provided it is done in a type safe way. Variables can be created inside blocks and are lexically scoped.

```
Bool ::= True | False
Num ::= 0, 1, 2, ... ∈ ℕ
Var ::= a, b, c, ... ∈ 𝕍
Expr ::= BoolExpr | NumExpr
Statement ::= skip
            | if BoolExpr
                then Statements
                else Statements
            | while BoolExpr
                Statements
            | Var := Expr
Statements ::= ϵ
             | Statement
               Statements

BoolExpr ::= Bool
           | Var
           | not BoolExpr
           | or BoolExpr BoolExpr
           | and BoolExpr BoolExpr
           | NumExpr < NumExpr
           | NumExpr > NumExpr
           | Expr == Expr
NumExpr ::= Num
          | Var
          | negate NumExpr
          | NumExpr + NumExpr
          | NumExpr - NumExpr
          | NumExpr * NumExpr
          | NumExpr / NumExpr
```

Fig. 1. The syntax of the While language

The syntax of the While language is shown in figure 1. Here the definition uses separate types for boolean and numeric expressions. This allows the definitions of monomorphic functions to be specified correctly (e.g. it is not possible to multiply a number by a boolean), but equality (`==`), which is polymorphic, is poorly specified. Typically at this point, the operational semantics would only define reduction where equality is used in a type-safe way, but we wish to be able to statically reject such incorrect use of equality as an invariant of the language rather than as a consequence of the operational semantics. Furthermore, we see that a Var can be either a numeric or a boolean value and can change type. We want the type system to reject programs that do unsafe things with variables.

One of the invariants we wish to enforce is that a variable can never be of an ambiguous type: that is, a preexisting variable can not change type within the body of the `while` loop (as the `while` loop may never be entered), and if it changes type within an `if` statement then both branches must result in the variable being the same type. This is a slightly stronger property than is absolutely necessary: it could be legal to change the type of a preexisting variable within the body of a `while` loop, or for an `if` statement to leave a variable with two types (or more, given nestings), provided that the variable is subsequently only assigned to and not read from. This would require a *def-use* analysis which is possible to implement with our technique, however, here we enforce only the simpler property.

With GADTs alone, you could implement many of these invariants statically. However, frequently the DSL would have to be extended to support

additional operations directly, for example adding explicit constructs to change the type of a variable, which somewhat goes against the notion of type *inference*, and additionally makes the DSL larger and more complex. In our opinion, much of the value of a DSL is in its ease of use (including type inference) and forcing the user to compromise on this to facilitate analyses is best avoided.

3 Static Analysis Framework

Our aim is to treat the type system as a programming environment as this will allow us to statically enforce the invariants as required. We shall rely on a library of functions that we have built, providing type level numbers, lists (which we frequently treat as sets) and maps and cover their APIs as necessary. Wherever possible we have tried to copy the APIs from normal value-level libraries such as Haskell's *Data.List* library in order to make these libraries as predictable and familiar as possible. In many cases, the only difference in use between our type-level versions and the normal value-level versions are that in using the type-level version you must put a type class constraint in the context of your function.

Our type level numbers consist of 11 simple data types: **data** $E = E$, which is a terminal, and 10 further data types representing each digit: **data** $D0\ n = D0\ n$, **data** $D1\ n = D1\ n$ and so forth. Thus the value $D4\ (D2\ E)$ has the type $D4\ (D2\ E)$ and when converted to an *Int* has the value 42. We define type classes so as to be able to increment and decrement these type level numbers (*Succ* and *Pred*), add them together, test for whether one is smaller than another, and the ability to convert to the corresponding value of type *Int*.

When checking a DSL-program, we need to be able to walk over the structure of the program, collecting information and checking that invariants are satisfied by each statement. To do this we ensure that the type of one statement depends upon the type of the preceding statement, thus analysis of the preceding statement must occur before the current statement can be typed. Hence we achieve type-level computation for each statement. These dependencies are created by using a type indexed monad: [3]

class *Monad* m **where**
$(\gg)\ \ :: m\ x\ y\ a \to m\ y\ z\ b \to m\ x\ z\ b$
$(\gg\!=)\ :: m\ x\ y\ a \to (a \to m\ y\ z\ b) \to m\ x\ z\ b$
$return :: a \to m\ x\ x\ a$

Thus the monad has gained two extra type parameters, the first can be thought of as the *from* state, and the second can be thought of as the *to* state. Now we can see that if we have statements of the While-DSL which are combined

[3] In this presentation we redefine the standard Haskell *Monad* type class as shown, which is legal Haskell 98 but isn't accepted by GHC prior to version 6.10. For versions of GHC prior to 6.10, we have to rename the monad functions and abandon **do**-syntax. Using **do**-syntax makes the presentation more familiar and simpler.

together using these monadic functions, then the type of the second statement must have a *from* state equal to the *to* state of the first statement, thus the dependency is achieved. It is into these type parameters that we shall place all the state that we require to check the properties of our DSL.

This monad can very easily be thought of as nothing more than the normal *State* monad except that the type of the state can change. The only instance of this monad that we need is similarly very close to the definition of the standard *State* data type in Haskell:

$$\begin{aligned}
&\textbf{newtype}\ \mathit{State}\ x\ y\ a = \mathit{State}\{\mathit{runState} :: x \to (a, y)\}\\
&\textbf{instance}\ \mathit{Monad}\ \mathit{State}\ \textbf{where}\\
&\quad f \gg g \quad = \mathit{State}\ (\lambda x \to \textbf{let}\ (_, y) = \mathit{runState}\ f\ x\ \textbf{in}\ \mathit{runState}\ g\ y)\\
&\quad f \gg\!= g \quad = \mathit{State}\ (\lambda x \to \textbf{let}\ (a, y) = \mathit{runState}\ f\ x\ \textbf{in}\ \mathit{runState}\ (g\ a)\ y)\\
&\quad \mathit{return}\ a = \mathit{State}\ (\lambda x \to (a, x))
\end{aligned}$$

So, in this framework, all we need is to define a data structure that will form the *from* and *to* states (i.e. the x and y parameters to the *State* type) and to define the functions that will allow the user to write a DSL-program.

For the While-DSL, we need to be able to identify variables uniquely at the type level in order to track their type and check their use. This requires that each new variable has a different type, and so we use type-level numbers to identify each variable. Thus the state must contain the next number to be used for the next variable. This shall be incremented as variables are created. Set membership is used for tracking the type of each variable. We need three sets to hold integer variables, boolean variables and newly created variables which have not yet been assigned to and thus have no type. The functions that we define which make up the While language will manipulate these sets and types in order to enforce the invariants as required.

Our framework can produce a value if the invariants of the DSL are satisfied, and for the While-DSL, we choose to produce an Abstract Syntax Tree (AST) of the program that has been supplied. This output is determined by the DSL-designer, who could equally decide no output is required and just use the framework for enforcing the DSL's invariants. We use a very simple ADT to produce the AST, one which does not enforce the invariants we are interested in. However, this ADT is never exposed to the user so is considered safe to use internally within the library. In this way, our framework acts as the bridge between the unsafe external world, statically rejecting bad DSL-programs, and the safe, controlled inner world of the library or DSL-designer. The ADT is shown in figure 2 and is very similar to the syntax of the While language in figure 1. From this ADT, it is straightforward to write an interpreter for the While-DSL which walks over the AST, if that is what the DSL-designer desires.

To create the AST, we need to be able to convert variables that are identified by types into Strings, i.e. when a variable is created, it will have a unique identifying type (a type-level number), e.g. $D5\ E$, but it will also have a name which is a String, for example `"f"`. Our type-level numbers support converting

```
type Statements
  = [Statement]
data Statement
  = Skip
  | Var := Expr
  | If BoolExpr Statements Statements
  | While BoolExpr Statements
    deriving (Show, Eq)
newtype Var
  = Var String
    deriving (Show, Eq)
data Expr
  = Boolean BoolExpr
  | Integer IntExpr
    deriving (Show, Eq)
```

```
data BoolExpr
  = BoolTerm Bool
  | BoolVar Var
  | Not BoolExpr
  | Or BoolExpr BoolExpr
  | And BoolExpr BoolExpr
  | Eq Expr Expr
  | LT IntExpr IntExpr
  | GT IntExpr IntExpr
    deriving (Show, Eq)
data IntExpr
  = IntTerm Int
  | IntVar Var
  | Negate IntExpr
  | IntExpr :+: IntExpr
  | IntExpr :*: IntExpr
  | IntExpr :/: IntExpr
  | IntExpr :-: IntExpr
    deriving (Show, Eq)
```

Fig. 2. The Algebraic Data Type of the While-DSL

a type-level number to an *Int*, so we have a *Map* from *Int*s to *String*s which will allow us to convert a variable as a type to its name for use in the AST, a list of *String*s to act as a source of value-level names for the variables, and a list of *Statement*s which will form the resulting AST of the While-program. So, as the While-DSL designer, we define the state as:

```
data DSLState nextVar unusedVars boolVars intVars
  = DS{ names      :: [String],
        nameMap    :: Map Int String,
        statements :: AST.Statements,
        nextVar    :: nextVar,
        unusedVars :: unusedVars,
        boolVars   :: boolVars,
        intVars    :: intVars }
```

and our functions will therefore be variations upon the type:

```
State (DSLState nextVar  unusedVars  boolVars  intVars)
      (DSLState nextVar' unusedVars' boolVars' intVars') result
```

permitting the types that hold the state we are interested in to vary as necessary. It is the relationships between these type parameters, enforced through type class contexts, which will allow the calculation of the *to* state from the *from* state, and will either accept or reject the provided While-program.

To run the framework, we need a function that will supply an initial state to the analysis. Considering the purpose of each component of the state,

the initial state is clear: the *nextVar* will be a type level number of zero (represented by *D0 E*), and the *unusedVars*, *boolVars* and *intVars* are all the empty list (*Nil*). We define the function *buildWithInvariants*, which applies this initial state to the While program supplied, and produces the AST as the output:

```
buildWithInvariants :: State (DSLState (D0 E) Nil Nil Nil)
                             (DSLState nextVar Nil boolVars intVars) () →
                        AST.Statements
buildWithInvariants prog = statements (snd (runState prog initialState))
  where
    initialState = DS{ names      = names,
                       nameMap    = Map.empty,
                       statements = [],
                       nextVar    = (D0 E),
                       unusedVars = Nil,
                       boolVars   = Nil,
                       intVars    = Nil}
    names = [[x] | x ← ['a'..'z']] ++
            [reverse (x : y) | y ← names, x ← ['a'..'z']]
```

The While language we are using does permit only the two types - integers and booleans - and so we have specialised our state with that in mind. If the language were to be extended such that new types could be declared, we would switch to using a type level map where the keys identify the types, and the corresponding values are lists of variables holding values of that type.

4 Type Inference

As the While-DSL designer and invariant enforcer, there are just five functions we need to implement: the four statements of the While-DSL and the ability to create new variables. The easiest of these is the `skip` statement which has no impact on typing whatsoever as it is a no-op. The implementation is very simple:

```
skip :: State (DSLState nextVar unusedVars boolVars intVars)
              (DSLState nextVar unusedVars boolVars intVars) ()
skip = State f where
  f ds@(DS{ statements }) = ((), ds{ statements = statements ++ [AST.Skip] })
```

Next we tackle creating a new variable. We need to increment the type *nextVar*, return and insert the old value into the *unusedVars* set, and create a corresponding value-level representation of the variable, as a *String*, shown in figure 3. *Num* is our module for type-level numbers and we see that to find the successor of a number is nothing more than putting the *Succ* type class in the context of our function. Similarly, with the *NumberToInt* type class (and corresponding *numberToInt* function), which converts a type-level number to an *Int*, and the *SetCons* type class which operates on type-level lists,

```
newVar :: ∀ nextVar nextVar' unusedVars unusedVars' boolVars intVars .
        (Num.Succ nextVar nextVar', NumberToInt nextVar,
         SetCons nextVar unusedVars unusedVars') ⇒
  State (DSLState nextVar  unusedVars  boolVars intVars)
        (DSLState nextVar' unusedVars' boolVars intVars) nextVar
newVar = State f
  where
    f ::  (DSLState nextVar unusedVars boolVars intVars) →
          (nextVar, (DSLState nextVar' unusedVars' boolVars intVars))
    f  ds@(DS{ names = (name : names), nameMap, nextVar, unusedVars})
       = (nextVar, ds{ names = names, nameMap = nameMap',
                       nextVar = nextVar', unusedVars = unusedVars'})
         where
           nameMap' = Map.insert varNum name nameMap
           nextVar' = Num.succ nextVar
           varNum = numberToInt nextVar
           unusedVars' = setCons nextVar unusedVars
```

Fig. 3. Creating a new variable

performing a *cons* as normal except when the element is already a member of the list, thus a list constructed in this way will not contain duplicates.

The three remaining functions (`:=`, `while` and `if`) all involve expressions so it is expressions we must tackle next. When a variable is used in an expression we do not know whether its current value is a boolean or integer, but the expression in which it is used demands that the variable's value is a particular type. Thus we need to check that the variable's current type, held in the type-level sets in the state, matches the use of the variable in the expression.

A straightforward expression ADT would give us a value of type *Expression* which would be useless to us as a type, as we need a much richer type to be able to verify that variables are being used correctly. Thus we use a GADT to build an expression such that the type of the expression contains the variables and the way in which they are used. Our idea is that the expression GADT has a type parameter which represents a stack, such that every element of the stack is a tuple, where the left of the tuple is the desired type, and the right of the tuple is a list of variables which are required to have the type on the left. For example:

Cons (*Int*, (*Cons X* (*Cons Y Nil*))) (*Cons* (*Bool*, *Cons Z Nil*) *Nil*)

is a type-level stack in which X and Y are variables which we need to be *Int*s and Z is a variable we need to be a *Bool*.[4] The expression GADT has three type parameters, the last of which is this stack structure, and the other two are nothing more than the decomposition of the tuple at the head of the

[4] *Cons* is the type-level equivalent of (:), just as *Nil* is the type-level equivalent of [].

```
data Expr ty vars stack where
  BC     :: Bool → Expr Bool Nil Nil
  IC     :: Int → Expr Int Nil Nil
  Var    :: (NumberToInt v) ⇒ v → Expr ty (Cons v Nil) Nil
  Not    :: Expr Bool vars stack → Expr Bool Nil (Cons (Bool, vars) stack)
  And    :: (Append vars vars' vars'', Append stack stack' stack'') ⇒
            Expr Bool vars stack → Expr Bool vars' stack' →
            Expr Bool Nil (Cons (Bool, vars'') stack'')
  Or     :: ...   -- as for And
  Eq     :: (Append vars vars' vars'', Append stack stack' stack'') ⇒
            Expr ty vars stack → Expr ty vars' stack' →
            Expr Bool Nil (Cons (ty, vars'') stack'')
  LT     :: (Append vars vars' vars'', Append stack stack' stack'') ⇒
            Expr Int vars stack → Expr Int vars' stack' →
            Expr Bool Nil (Cons (Int, vars'') stack'')
  GT     :: ...   -- as for LT
  Negate :: Expr Int vars stack → Expr Int Nil (Cons (Int, vars) stack)
  Plus   :: (Append vars vars' vars'', Append stack stack' stack'') ⇒
            Expr Int vars stack → Expr Int vars' stack' →
            Expr Int Nil (Cons (Int, vars'') stack'')
  Minus, Times, Divide :: ...   -- as for Plus
```

Fig. 4. The *Expr* GADT

stack. This simplifies construction of the stack. The *Expr* GADT is shown in figure 4. [5]

The constructors for constants (*BC* and *IC*), being leaves of the expression tree, have an empty stack and do not use variables. Using a variable is achieved with the *Var* constructor, which is also a leaf of the expression tree so we construct just a singleton list of the variable, but we don't know at this point the type which the variable is expected to have, thus we leave the type floating as *ty*.

The *And* constructor takes two sub-expressions which must both produce *Bool*s, we combine both the subexpressions' lists of variables, and their stacks. Thus in the expression *And* (*Var a*) (*Not* (*Var b*)) the variable *a* would be in the *vars* type parameter for the left child of *And*, so would be placed on the top of the stack by the *And* constructor, whereas the variable *b* would already be in the stack, having been placed there by the *Not* constructor. Thus this GADT only applies types to variables which are direct children of any given constructor, which is what we, as the DSL-designer, require.

Eq is polymorphic, merely requiring the subexpressions are the same type. The result of equality is a *Bool*; and when building the stack, we use the com-

[5] *Append* is a type-level version of (++).

mon *ty* type variable of the subexpressions. This means that the expression *Eq* (*Var a*) (*Var b*) does not have a ground type as the type of the variables *a* and *b* are not fully known: they could both be *Bool*s or *Int*s. Note that in capturing the use of variables within this expression sub-language, we have shown that our technique is applicable to both statement and expression-oriented languages.

Type checking is then quite simple as we only need to walk over the stack and check that the variables used are members of the correct sets of variables:

class *ValidVarUse stack boolVars intVars*

instance *ValidVarUse Nil boolVars intVars*

instance (*ValidVarUse stack boolVars intVars*,
IsSubList (*Cons v vars*) *intVars isInt*,
IsSubList (*Cons v vars*) *boolVars isBool*,
Bool.Not isInt isBool, *Bool.If isBool Bool Int ty*) ⇒
ValidVarUse (*Cons* (*ty*, (*Cons v vars*)) *stack*) *boolVars intVars*

instance (*ValidVarUse varStack boolVars intVars*) ⇒
ValidVarUse (*Cons* (*ty*, *Nil*) *varStack*) *boolVars intVars*

We recurse over the stack, and for each element we have two cases, for when variables have and have not been used. The lack of a ground type for certain expressions means that we can't match *Bool* or *Int* on the type parameter *ty* at the head of the stack as *ty* may be unknown. Instead, we reason that the variables must be a sublist of either the booleans or the integers *but not both* (achieved through the *Not* relation), and then we require that if they are a sublist of the booleans then *ty* must be *Bool* otherwise it must be *Int*; the *Bool.If c t f r* class context should be read as $r =$ **if** c **then** t **else** f. *IsSubList* and the *Bool* module are both provided by our libraries of type-level functions.

For assignment we need one further type class (called *TypeVar*) which updates the sets of variables correctly. The variable being assigned to is removed from its current set and inserted into the set corresponding to the type of the expression. Sadly, this can't be implemented quite this simply as it is a challenge to make GHC understand that removing an element from a set and then adding it straight back in results in the same original set (a property we need for showing that the body of a `while` loop does not alter the type of any variables), thus we have to do some set membership tests first. The type class is otherwise straightforward. Assignment then checks that the expression uses variables correctly (the *ValidVarUse* type class), it updates the type of the variable being assigned to (the *TypeVar* type class), and it suitably adds to the list of *Statement*s for the AST (which requires the *NumberToInt* type class and also the *NumberListToIntList* type class which is a mapping of the former over a list of type-level numbers). The conversion from the *Expr* GADT to the AST is entirely mechanical. The type of assignment is therefore:

(.=) :: (*TypeVar var ty unusedVars boolVars intVars*

```
                                unusedVars' boolVars' intVars',
        ValidVarUse (Cons (ty, vars) stack) boolVars intVars,
        NumberToInt var,
        NumberListToIntList boolVars, NumberListToIntList intVars) ⇒
      var → Expr ty vars stack →
      State (DSLState nextVar unusedVars  boolVars  intVars)
            (DSLState nextVar unusedVars' boolVars' intVars') ()
var .= expr = ...
```

The `if` and `while` statements are very similar so we only present the `while` statement as it is the simpler of the two. The expression is type checked as for assignment. The body may create lexically scoped variables, so variables that are created within the body must not escape into the outer scope. We create a type class *RemoveVars* which takes the range of variables created (i.e. *nextVar* to *nextVar'*) and removes variables within that range from the sets of variables. Finally, we require that after removing variables created within the body, the body results in the same sets of variables with the same type as were initially supplied. Thus the body cannot leave a variable in a different type than it was before the `while` statement. The body of the function is not enormously interesting, whereas the type is.

```
while :: (ValidVarUse (Cons (Bool, vars) stack) boolVars intVars,
          RemoveVars nextVar nextVar' unusedVars' boolVars' intVars'
                                      unusedVars  boolVars  intVars,
          NumberListToIntList boolVars, NumberListToIntList intVars) ⇒
  Expr Bool vars stack →
  (State (DSLState nextVar  unusedVars  boolVars  intVars)
         (DSLState nextVar' unusedVars' boolVars' intVars') ()) →
  (State (DSLState nextVar  unusedVars  boolVars  intVars)
         (DSLState nextVar' unusedVars  boolVars  intVars)  ())
while cond body = ...
```

In Haskell, **if** is a keyword so cannot be redefined, so we instead call the function if_w. This function uses all the same machinery, just slightly differently. The two bodies of the `if` statement can leave variables with different types than before the `if` statement, provided they both make the same changes.

And that is it: in under 400 lines of Haskell specific to the While-DSL, we can now write While programs, supply them to the *buildWithInvariants* function, and if no error occurs, know that the invariants of the language were statically checked and a simple AST of the program is returned. The code is not particularly complex, and whilst it is more verbose than equivalent runtime checks, the advantages of statically asserting these invariants more than outweighs the extra code size.

5 Extending While with Security

To demonstrate how flexible and extensible this framework is, we extend the While-DSL with the Bell / LaPadula security model [3], which constrains information flow. Variables are given security categories such as *public* or *top-secret*, which are totally-ordered, and the system ensures that no *read-ups* or *write-downs* occur. These can happen in assignments where the expression on the right of the assignment reads a variable which has a higher security level (so a *read-up*) than the variable to which the expression is being assigned (and so represents an information leak) and viewing it the other way around, it is also a *write-down* as privileged information is being written down to a lower level. But it doesn't just occur in assignments: the condition of a `while` or `if` statement imposes a minimum security level on the bodies of those statements for the very fact that a body is being executed reveals information about the conditional. Thus every statement within the body must be at the same security level or higher than is required to read the variables within the conditional expression. This analysis could, for example, be used to ensure that data read from a database is never directly returned to a user, or that passwords are never held by user-visible variables.

Our implementation extends this further: when a variable is created, we provide two security levels, a minimum and a maximum. These can be the same value, in which case checking is performed as described above. However, the minimum value can be less than the maximum and indicates that we're not sure what the security level of the variable should be. The maximum level will not change, and so if the program is rejected then we know that some variable is not sufficiently privileged given the maximum security levels, but the minimum level can rise, which indicates that the variable must be granted at least the resulting minimum level for the program to be acceptable. The maximum level is used when reading a variable, and the minimum level is used when writing to a variable. Thus raising the minimum security level is always a safe operation, because it only increases what can be written to a variable but does not alter who can read that variable. So raising the minimum level of a variable cannot invalidate assignments or conditionals that have already been checked. Our security levels are type-level numbers, where zero (*D0 E*) is the lowest possible level.

To implement this security analysis, we need to extend the type-level state. We need to carry around a minimum security level: this is set by `if` and `while` conditionals and represents the minimum security level a variable must have to be assigned to within such a statement. We also need a type-level map (provided by our *TMap* module), from variables to tuples representing the current minimum and maximum security level. The `skip` statement has no security implications and so is unaltered. Creating a new variable is as before, except it inserts the minimum-maximum tuple into the security map, and we check the minimum is less than the maximum. Expressions already expose in

their type all the variables that are used, so we once again need to be able to walk this structure and extract the highest maximum security level, which we will use as the *read level* of the expression. This is the purpose of the *MaxSecurityLevel* type class:

```
class MaxSecurityLevel map varStack maxIn maxOut
        | map varStack maxIn → maxOut where
  maxSecurityLevel :: map → varStack → maxIn → maxOut
```

The instances of this type class walk first over the stack of tuples and then over the inner lists of variables, indexing the *map* and testing to see if the maximum security level of each variable is greater than the current security level (held by *maxIn*). The maximum level found is returned in *maxOut*.

Assignment then uses the *MaxSecurityLevel*, finding the *read level* of the expression (*exprMax*). We check to see if this is lower than the minimum level currently held by the state (*exprMin*), if so, the state's level is used it its place. This ensures that if the conditional of an `if` or `while` statement uses variables at a particular level (and hence updates the state to hold that level) then assignments within the bodies of those statements must be valid for at least the same security level, if not higher. The variable to which we are assigning must have a minimum level greater than or equal to the *read level*. If this is not the case then we attempt to raise its minimum level sufficiently, but enforce that this must still remain below its maximum level. Finally we update the security map (*sMap*) with the new security levels. The additional type class contexts are shown below in black and the existing previous code in grey:

```
(.=)::(TypeVar var ty unused bools ints unused' bools' ints',
       ValidVarUse (Cons (ty, vars) stack) bools ints, NumberToInt var,
       NumberListToIntList bool, NumberListToIntList ints,
       MaxSecurityLevel sMap (Cons (ty, vars) stack) (D0 E) exprMax,
       IsSmallerThan exprMax exprMin exprMaxSmallerThanMin,
       Bool.If exprMaxSmallerThanMin exprMin exprMax exprMax',
       TMap.Lookup sMap var (varMin, varMax),
       IsSmallerThan varMin exprMax' varMinNeedsRaising,
       Bool.If varMinNeedsRaising exprMax' varMin varMin',
       SmallerThanOrEq varMin' varMax,
       TMap.UpdateVarying sMap var (varMin', varMax) sMap') ⇒
  var → Expr ty vars stack →
  State (DSLState nextVar unused bools ints sMap exprMin)
        (DSLState nextVar unused' bools' ints' sMap' exprMin) ()
var .= expr = ...
```

The `if` and `while` statements are again very similar. They extract the *read level* of the conditional, *exprMax*, find the maximum of that value and the level currently held by the state (to ensure the minimum level isn't decreased by nested statements), *exprMin'*, and update the state used for their bodies with this value. Thus the final security map, *sMap'*, contains the security

requirements of the bodies, which were built using the updated minimum expression level.

```
while::(ValidVarUse (Cons (Bool, vars) stack) bools ints,
        RemoveVars nextVar nextVar' unused' bools' ints' unused bools ints,
        NumberListToIntList boolVars, NumberListToIntList intVars,
        MaxSecurityLevel sMap (Cons (Bool, vars) stack) (D0 E) exprMax,
        IsSmallerThan exprMin exprMax minExprNeedsRaising,
        Bool.If minExprNeedsRaising exprMax exprMin exprMin') ⇒
  Expr Bool vars stack →
  (State (DSLState nextVar unused bools ints sMap exprMin')
         (DSLState nextVar' unused' bools' ints' sMap' exprMin'') ()) →
  (State (DSLState nextVar unused bools ints sMap exprMin)
         (DSLState nextVar' unused bools ints sMap' exprMin) ())
while cond body = ...
```

Finally, we alter the *buildWithInvariants* function so that it returns not only the AST of the While program, but also the security map, converted to a normal *Data.Map* value. Thus adding just two further parameters to the *DSLState*, and a few additional type class contexts where necessary, has allowed us to extend the type inference machinery to enforce a powerful and useful security model.

6 Related Work

This work arose out of the irritation of using DSLs without sufficient static checking. Although we have developed our own type-level libraries, most of the functionality and design is very similar to previous type-level programming work [4].

It has been shown in [5] that by using Phantom Types (which we use extensively of in this work through GADTs), a monomorphic higher-order language can be embedded into Haskell in such a way that type soundness and completeness are achieved: i.e. an ill-typed program in the monomorphic higher-order language cannot be represented by the embedding and a well-typed program in the monomorphic higher-order language can always be represented by the embedding. Whilst our work offers no formal proofs, we do cater for polymorphic functions (equality), linear type inference (variables can change type), and work with an imperative language that is further away from Haskell than the λ-calculus they use.

Lava [6] is a DSL embedded in Haskell which caters for circuit design with support for simulation and verification of electronic circuits, in addition to traditional hardware design tools. The library takes advantage of Haskell's type classes to permit easy extension of the library and to enforce type safety. However, the basic type used is a representation of a bit or a number. Desired circuit invariants are supported through its verification mechanisms which makes use of external theorem provers. It would be very interesting to explore

how our framework could allow circuit designers to enforce invariants of their circuits without resorting to external theorem provers and what the advantages and disadvantages of each approach are.

Modelling security properties within languages is a well studied field and has received attention in Haskell as well as other languages. In [7], a security model is presented in the form of a typical library for Haskell programs rather than for embedded DSLs. They develop declassification policies in which previously secret data can be declassified and made available at a lower security level. In [8], the Dependency Core Calculus (DCC) of [9] is successfully translated into System-F and a Haskell implementation provided for a 2-point lattice. The DCC allows for modelling program analyses which feature dependencies throughout programs, broader than just security analysis. [8] demonstrates that the dependencies captured by the DCC can be similarly captured by the parametricity theorem for System-F.

In [10], a dependently typed language is developed and used to stage DSLs. The focus is to eliminate unnecessary tagging operations that typically arise when the guest DSL language and the host language are both statically typed. A formalisation is presented that proves type safety of the host language, MetaD, and its staging constructs.

7 Conclusions and Future Work

The convenience and power of embedding DSLs within a language make them an attractive and widely used technique. Statically enforcing their invariants increases this appeal as it makes using a DSL more robust, reduces unexpected surprises for Haskell programmers, and makes designing and implementing a DSL less error prone. We have presented a flexible technique for lifting the invariants of a DSL into the Haskell type system and demonstrated how our framework can be used to statically enforce invariants relating to type checking and inference, and the Bell / LaPadula security model. Whilst the While language used is a simple statement based language with an expression sub-language, we believe the Haskell type system is capable of capturing many invariants from many different styles of DSLs.

Adding the Bell / LaPadula security model extended the existing code in place. However, by using type classes for the operations of the While language, a more modular means of extension is possible. With a careful design of the state, both run-time and compile-time code can be reused, allowing abstraction and encapsulation.

Although we show how errors can be caught statically, the error messages are expressed in terms of the underlying mechanics of the library, and are not likely to be of great use to the user (unless they understand how the analysis works). Future work is looking at a range of ideas to allow more meaningful error messages to be produced, e.g. by tracking the error state explicitly through the type system, though this makes the type-level machinery more

complex.

The analyses presented here only require that the type of each statement is dependant on the type of the preceding statement. Analyses which reach fixed points (for example, *live-variable* analysis), require that the type of a statement depends on the types of all preceding (or succeeding) statements as defined by the control flow graph rather than just the list of statements. This control-flow graph case is more complex, though we believe it is possible with our technique. With the right options, GHC's type checker is Turing-complete. This should mean that any general purpose analysis can be used on an DSL. However, for large DSL programs, this may prove infeasible as type checking may become very slow, or checking the DSL program could fail to terminate.

References

[1] Peyton Jones, S.L., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. In: ICFP. (2006) 50–61

[2] Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer Verlag (1999)

[3] Bell, D.E., Lapadula, L.J.: Secure computer systems: Mathematical foundations (1973)

[4] Kiselyov, O., Lämmel, R., Schupke, K.: Strongly typed heterogeneous collections. In: Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell, ACM Press (2004) 96–107

[5] Rhiger, M.: A foundation for embedded languages. ACM Trans. Program. Lang. Syst. **25**(3) (2003) 291–315

[6] Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: hardware design in haskell. In: ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming, New York, NY, USA, ACM Press (1998) 174–184

[7] Russo, A., Claessen, K., Hughes, J.: A library for light-weight information-flow security in haskell. In: Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell, New York, NY, USA, ACM (2008) 13–24

[8] Tse, S., Zdancewic, S.: Translating dependency into parametricity. In: ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming, New York, NY, USA, ACM (2004) 115–125

[9] Abadi, M., Banerjee, A., Heintze, N., Riecke, J.G.: A core calculus of dependency. In: POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (1999) 147–160

[10] Pašalić, E., Taha, W., Sheard, T.: Tagless staged interpreters for typed languages. SIGPLAN Not. **37**(9) (2002) 218–229

A Pure Object-Oriented Embedding of Attribute Grammars

Anthony M. Sloane [1,2] Lennart C. L. Kats, Eelco Visser [1,3]

Department of Computing
Macquarie University
Sydney, Australia

Software Engineering Research Group
Delft University of Technology
Delft, The Netherlands

Abstract

Attribute grammars are a powerful specification paradigm for many language processing tasks, particularly semantic analysis of programming languages. Recent attribute grammar systems use dynamic scheduling algorithms to evaluate attributes by need. In this paper, we show how to remove the need for a generator, by embedding a dynamic approach in a modern, object-oriented programming language to implement a small, lightweight attribute grammar library. The Kiama attribution library has similar features to current generators, including cached, uncached, circular, higher-order and parameterised attributes, and implements new techniques for dynamic extension and variation of attribute equations. We use the Scala programming language because of its combination of object-oriented and functional features, support for domain-specific notations and emphasis on scalability. Unlike generators with specialised notation, Kiama attribute grammars use standard Scala notations such as pattern-matching functions for equations and mixins for composition. A performance analysis shows that our approach is practical for realistic language processing.

Key words: language processing, compilers, domain-specific languages

1 Introduction

The language processing domain concerns the construction of compilers, interpreters, code generators, domain-specific language implementations, refactoring tools, static code analysers and other similar artefacts. Attribute grammars are a powerful processing formalism for many tasks within this domain, particularly for semantic analysis of programming languages [7,22].

Attribute grammars extend context-free grammars with declarative equations that relate the values of attributes of grammar symbols to each other. Most attribute grammar systems translate the equations into an implementation written in

[1] This research was supported by NWO projects 638.001.610, *MoDSE: Model-Driven Software Evolution*, 612.063.512, *TFA: Transformations for Abstractions*, and 040.11.001, *Combining Attribute Grammars and Term Rewriting for Programming Abstractions*.

[2] Email: `Anthony.Sloane@mq.edu.au`

[3] Email: `L.C.L.Kats@tudelft.nl`, `visser@acm.org`

a general purpose programming language. The translation makes decisions about attribute evaluation order and storage, removing the need for manual techniques such as visitors. Therefore, the developer is freed to focus on the language properties that are represented by the attributes.

In recent years, attribute grammar systems have focused on dynamically scheduled evaluation, where the attributes to be evaluated and the evaluation order are determined at run-time rather than at generation time [14]. LRC [23], JastAdd [11], UU AG [2], and Silver [25] are prominent examples of this approach. A dynamic schedule has the advantage that attributes are evaluated at most once, but adds run-time overhead. In applications such as integrated development environments, the tradeoff is particularly worthwhile, since not all attributes are needed at all times.

Nevertheless, these recent systems are based on generators that add to the learning curve and complicate the development and build processes. We show in this paper how to integrate a dynamically scheduled attribute grammar approach as a library into an existing modern, object-oriented language. We use a *pure embedding* where the syntax, concepts, expressiveness and libraries of the base language are used directly [12,18]. The high-level declarative nature of the attribute grammar formalism is retained and augmented with the flexibility and familiarity of the base language, both for specification and for implementation of the formalism itself.

This work is part of the Kiama project [24] [1] that is investigating pure embedding of language processing formalisms into the Scala programming language [20]. The main reasons for using Scala are its inclusion of both object-oriented programming and functional programming features, support for domain-specific notations, emphasis on scalability and interoperability with the Java virtual machine.

Kiama's attribution library has the same general power as systems such as JastAdd [11]. [2] Abstract syntax trees are defined by standard Scala classes with only minimal augmentation of the class definitions required to prepare them for attribution. Attribute equations are written as pattern matching functions of abstract tree nodes. As well as basic synthesised and inherited non-circular attributes, Kiama currently supports reference attributes [10], higher-order or non-terminal attributes [26], parameterised attributes [8], and circular attributes that are evaluated to a fixed point [16]. Language extension and modification are achieved using Scala's scalability constructs such as traits and mixins. Also, in contrast to previous systems, attribute definitions can be adapted at run-time to implement dynamic language variations. Overall, the performance of Kiama attribute evaluators is similar to dynamically scheduled evaluators produced by generators.

The rest of the paper is structured as follows. Section 2 provides an introduction to the features of Kiama's attribution library by way of two typical examples. The Kiama implementation is outlined in Section 3. Section 4 considers how language extension and separation of concerns can be achieved by leveraging the general Scala platform. We evaluate the performance of Kiama in Section 5. The paper

[1] http://plrg.science.mq.edu.au/projects/show/kiama

[2] Like JastAdd, Kiama also has facilities for tree rewriting, but they are beyond the scope of this paper.

concludes with a discussion of our approach in the context of other attribute grammar systems in Section 6 and concluding remarks in Section 7.

2 Attribute Grammars in Kiama

This section presents a couple of well-known examples to introduce the basic capabilities and style of the Kiama attribution library.

2.1 Repmin

Repmin is a classic problem of tree analysis and transformation, originally employed to illustrate the use of lazy circular programs in functional programming to eliminate multiple tree traversals [3]. Repmin is often used as a simple test of attribute grammar systems. The problem is to take a binary tree with integer leaves and transform it into a tree with the same structure, but with each leaf value replaced by the minimum leaf value of the original tree.

Kiama is intended to work as seamlessly as possible with a developer's non-Kiama Scala code, including libraries. Attribution is performed on trees made from standard Scala *case class* instances. A case class allows instances to be created without the usual `new` operator, provides structural equality and supports structure-based pattern matching. In this sense, case classes provide capabilities that are similar to algebraic data types found in languages such as Haskell and ML.

Figure 1(a) shows the abstract syntax for Repmin in Scala and a typical problem instance. Each class inherits from the `Attributable` Kiama library class to obtain generic functionality, but otherwise no changes are necessary. The case classes can have other fields, members, supertypes and so on, without affecting the attribution.

Figure 1(b) shows the definitions of the `locmin` (local minimum), and `globmin` (global minimum) integer-valued attributes and the `repmin` tree-valued attribute. (In this example, no attributes of `repmin` are demanded, but they could be, making it a higher-order attribute.) `attr` is a Kiama library function that takes as argument the attribute equations defined by cases on the node type. Each resulting attribute is a function from a node type to the type of the attribute value. Attributes in modular specifications should be partial functions to allow for composition, so Kiama constructs the type of an attribute using its own `==>` partial function type constructor instead of the usual Scala (total) function type constructor `=>`.

The pattern matching abilities of case classes are used in the attribute equations. Identifiers beginning with a lowercase letter are binding occurrences, whereas those beginning with an uppercase letter are constants. An underscore pattern matches anything. A `v @ p` pattern binds the name `v` to the value matched by the pattern `p`. A guard `if boolexp` matches if the expression `boolexp` evaluates to true.

On the right-hand side of an equation, attributes are accessed using a reference style: the value of attribute `a` of node `n` is written `n->a`. The definition of `globmin` uses pre-defined *structural properties* to inspect the tree structure: `t isRoot` is true if `t` is the root of the tree and `t.parent` is a reference to `t`'s parent. (Scala allows

```
abstract class Tree extends Attributable
case class Pair (left : Tree, right : Tree) extends Tree
case class Leaf (value : Int) extends Tree

// repmin (Pair (Leaf (3), Pair (Leaf (1), Leaf (10))))
//       == Pair (Leaf (1), Pair (Leaf (1), Leaf (1)))
```

(a) Scala abstract syntax for Repmin trees and a simple problem instance.

```
val locmin : Tree ==> Int =
    attr {
        case Pair (l, r) => (l->locmin) min (r->locmin)
        case Leaf (v)    => v
    }

val globmin : Tree ==> Int =
    attr {
        case t if t isRoot => t->locmin
        case t             => t.parent[Tree]->globmin
    }

val repmin : Tree ==> Tree =
    attr {
        case Pair (l, r)  => Pair (l->repmin, r->repmin)
        case t @ Leaf (_) => Leaf (t->globmin)
    }
```

(b) Kiama attribute grammar for Repmin.

Fig. 1. A Kiama solution to the Repmin problem.

the period in a method call `o.m` to be omitted, so `t isRoot` is just `t.isRoot`, and similarly for the call of the `min` method.) Note that since the parent has a generic type, it must be cast to a `Tree`. Section 6 revisits the typing question.

Overall, Repmin is defined in Kiama in a clear and natural way using mostly standard Scala features. Specialising an equation for a particular node type is easy using pattern matching. Defining more complex grouping of attribution is also straight-forward. For example, the definition of `globmin` applies at all nodes and propagates the root value down the tree in a modular fashion, without requiring voluminous copy rules or special constructs as in some other systems.

2.2 *Variable liveness*

Attribute grammars were originally designed to express computations on tree structures. With the addition of remote node references that follow naturally from an object-oriented representation of attribute grammars, graph-based algorithms can also be expressed [10]. For instance, references allow attributes to define a control flow graph. Furthermore, using fixed point iteration to evaluate attributes, attribute grammars can be used to express data-flow equations [16]. We illustrate these capabilities using a variable liveness computation for a simple imperative language [19].

Figure 2(a) shows a typical variable liveness problem instance, where the *In* and *Out* sets are the live variables reaching or leaving each statement. Figure 2(b) shows the abstract syntax that is used for this example. In the definition of the

```
                          In          Out
{
    y = v          {v, w}      {v, w, y}
    z = y          {v, w, y}   {v, w}
    x = v          {v, w}      {v, w, x}
    while (x) {    {v, w, x}   {v, w, x}
        x = w      {v, w}      {v, w}
        x = v      {v, w}      {v, w, x}
    }
    return x       {x}
}
```

(a) A variable liveness problem instance.

```
type Var = String
abstract class Stm extends Attributable
case class Assign (left : Var, right : Var) extends Stm
case class While (cond : Var, body : Stm) extends Stm
case class If (cond : Var, tru : Stm, fls : Stm) extends Stm
case class Block (stms : Stm*) extends Stm
case class Return (ret : Var) extends Stm
case class Empty () extends Stm
```

(b) Scala abstract syntax definition for liveness problem.

Fig. 2. The variable liveness problem.

`Block` class, the type `Stm*` is standard Scala that indicates that the `stms` field is a sequence of zero or more statements, implemented by the Scala collection library.

The liveness sets for a statement s are calculated from the variables defined by $s(defines)$ and the variables used by $s(uses)$ by iterative application of the standard data flow equations $in(s) = uses(s) \cup (out(s) \setminus defines(s))$ and $out(s) = \bigcup_{x \in succ(s)} in(x)$, where $succ(s)$ denotes the control-flow successors of s.

Figure 3 shows the Kiama definitions of these attributes. The control flow successor `succ` is a reference attribute defined in terms of a `following` attribute that defines the default linear control flow. `following` is defined as an inherited attribute by pattern matching on the parent node, using the convenience function `childAttr`. The `_*` patterns in these definitions match possibly-empty sequences.

The `circular` Kiama library function used in the definitions of in and out is like `attr` except that it also takes an initial value for the attribute and evaluates until a fixed point is reached. An alternative attribute access notation `a (n)` has been used for the liveness sets to emphasise the correspondence with the data flow equations. In the definition of out, the Scala library method `flatMap` applies in to each of the statement's successors and concatenates the results.

As in the Repmin example, the variable liveness definitions are relatively easy to follow, use mostly standard Scala, and correspond closely to the mathematical definitions of the various properties. The small Kiama attribution library interface is summarised in Figure 4. The next section outlines its implementation.

```
val succ : Stm ==> Set[Stm] =
    attr {
        case If (_, s1, s2)   => Set (s1, s2)
        case t @ While (_, s) => t->following + s
        case Return (_)       => Set ()
        case Block (s, _*)    => Set (s)
        case s                => s->following
    }

val following : Stm ==> Set[Stm] =
    childAttr {
        case s => {
            case t @ While (_, _)           => Set (t)
            case b @ Block (_*) if s isLast => b->following
            case Block (_*)                 => Set (s.next)
            case _                          => Set ()
        }
    }

val uses : Stm ==> Set[String] =
    attr {
        case If (v, _, _)  => Set (v)
        case While (v, _)  => Set (v)
        case Assign (_, v) => Set (v)
        case Return (v)    => Set (v)
        case _             => Set ()
    }

val defines : Stm ==> Set[String] =
    attr {
        case Assign (v, _) => Set (v)
        case _             => Set ()
    }

val in : Stm ==> Set[String] =
    circular (Set[String]()) {
        case s => uses (s) ++ (out (s) -- defines (s))
    }

val out : Stm ==> Set[String] =
    circular (Set[String]()) {
        case s => (s->succ) flatMap (in)
    }
```

Fig. 3. Kiama attribute grammar for the variable liveness problem.

3 Implementation

The Kiama implementation consists of two main parts: definitions of structural properties and evaluation mechanisms for the different kinds of attribute. The implementation of Kiama consists of about 230 lines of Scala code.

3.1 Structural Properties

Case classes to be attributed must inherit from the `Attributable` trait. Each case class is automatically an instance of Scala's `Product` trait that provides generic access to its constructor fields. The code that initialises an `Attributable` instance uses the `Product` interface to set the structural properties, such as `parent`, and, for nodes in sequences, `next` and `prev`.

A complication is that the attribution library must coexist with Scala code that processes the same data structures. In particular, nodes might contain sequences

`Attributable`	Supertype of all node types.
Structural attributes of all nodes	
`t.parent : Attributable`	Parent of `t`.
`t.isRoot : Boolean`	Is `t` the root of the tree?
Structural attributes of nodes occurring in sequences of nodes of type `T`	
`t.prev, t.next : T`	Siblings of `t`.
`t.isFirst, t.isLast : Boolean`	Is `t` the first or last node?
`t.index : Int`	Number of siblings before `t`.
For node type `T`, user-defined attributes of type `U`	
`attr (f : T => U) : T ==> U`	Basic attribute defined by `f`.
`circular (init : U) (f : T => U) : T ==> U`	Circular attribute defined by `f` with initial value `init`.
`childAttr (f : T => Attributable ==> U) : T ==> U`	Attribute defined by matching on parent.
`paramAttr (f : S => T ==> U) : S => T ==> U`	Attribute with parameter of type `S`.
Access attribute `a` of node `n`	
`n->a`	Reference style.
`a (n)`	Functional style.

Fig. 4. Summary of the Kiama attribution interface.

```
case class Upper (a : Lower, b : Lower*, c : Int,
                  d : Option[Lower]) extends Attributable
case class Lower (...) extends Attributable
```

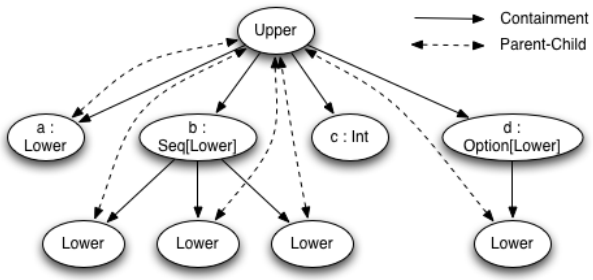


Fig. 5. The Kiama parent-child relation compared to structure containment.

and optional fields represented by Scala values of type `Seq[T]` and `Option[T]`. (`Option[T]` is analogous to Haskell's `Maybe a` type, having values of `None` or `Some (t)`, for some value `t` of type `T`.) Fields that are not attributable might also be present, most notably primitive values.

To address these issues, Kiama makes a distinction between the *containment relation* between a node and its fields as defined by the case class declaration, and the *parent-child relation* that relates an `Attributable` node to its `Attributable`

```
class CachedAttribute[T,U] (f : T ==> U) extends (T ==> U) {
    val memo = new IdentityHashMap[T,Option[U]]

    def apply (t : t) : U =
        memo.get (t) match {
            case None             => memo (t) = None
                                     val u = f (t)
                                     memo (t) = Some (u)
                                     u
            case Some (Some (u)) => u
            case Some (None)     => error ("Cycle detected")
        }

    def isDefinedAt (t : T) : Boolean = f isDefinedAt t
}
```

Fig. 6. The CachedAttribute class.

children. Both of these relations are useful in attribute equations. Figure 5 shows an example where an `Upper` node contains four fields: one required `Lower`, a sequence of zero or more `Lower` nodes, an integer and an optional `Lower`. The `Upper` node therefore has five `Attributable` children and those nodes have the `Upper` node as their parent. Most accesses to nodes in equations are performed via fields or the `parent` property, but Kiama also provides an iterator so that all `Attributable` children can be accessed in a generic way.

3.2 Attributes

Attributes defined by `attr` are implemented by the `CachedAttribute` class that we focus on here. Since attribute equations are cached and are not evaluated until they are needed, the evaluation method is equivalent to those used in early attribute grammar systems [14] and, more recently, in JastAdd [11].

Figure 6 shows the definition of the `CachedAttribute` class. The type parameters `T` and `U` denote the type of the nodes to which this attribute applies and the type of the attribute value, respectively. The value parameter `f` is the user-specified (partial) function that defines the attribute equations. Since `CachedAttribute` is a sub-class of the partial function type `T ==> U` and Scala converts `a (n)` into `a.apply (n)`, this implementation presents a convenient functional interface to the attribute value. The reference notation `n->a` is a simple alias.

The partial function implementing an attribute must define two methods: `apply`, that "runs" the defining equations on the given node and returns the value, and `isDefinedAt`, that provides information about the function's domain. For a cached attribute, `apply` uses a local hash map to memoise the attribute value for the node `t`. A marker value `None` is used to detect when the method calls itself, so that an error can be reported. The `isDefinedAt` method simply delegates to the `isDefinedAt` of the attribute equations.

Other kinds of attributes are defined by similar classes with the same interface. For example, uncached attributes are a simple variant. `circular` uses a `CircularAttribute` class that provides a functional interface to the fixed-point evaluation algorithms of Magnusson and Hedin [16].

It is sometimes useful to have attributes that are parameterised by other values. For example, JastAdd specifications often use this style in name analysis where a `lookup` attribute is parameterised by the name being sought [8]. Parameterised attributes are created in Kiama using the `paramAttr` function (Figure 4). For example, an attribute for looking up a name `n` can be defined in Kiama as follows.

```
val lookup : String => Attributable ==> Decl =
    paramAttr {
        n => {
            case ...  // cases for lookups at different nodes
        }
    }
```

4 Language Extensions and Separation of Concerns

Many attribute grammar systems allow for a high degree of separation of concerns, allowing different equations for an attribute or production to be defined across different modules. Typically, this modularity is implemented as a purely syntactic feature, joining together all equations for an attribute before compilation, and considering the entire, merged specification as a whole. (This is notably not the case for first-class attribute grammars [5], where attributes are *first-class citizens* and can be manipulated in the language.)

While other attribute grammar systems often use a general-purpose language for the expressions in attribute equations (e.g., Haskell in UU AG [2], Java in JastAdd), they provide their own module systems on top of that language. Kiama relies purely on Scala for the modular specification of attribute grammars. As a modern object-oriented programming language aimed at high-level abstraction for building modular frameworks with a rich, often functional interface, Scala offers an impressive toolbox of modularization features, most notably traits and mixins.

4.1 Static Separation of Concerns Using Traits

Flexible static combination of attribution modules can be achieved using Scala traits to define components and performing mixin composition to combine them [21]. For example, we can decompose the variable liveness problem of Section 2.2 into three components dealing with control flow, variables, and the liveness computation itself. The first two of these can be abstracted by interfaces defined by traits.

```
trait ControlFlow {
    val succ : Stm ==> Set[Stm]
}

trait Variables {
    val uses : Stm ==> Set[String]
    val defines : Stm ==> Set[String]
}
```

An implementation of the liveness module can use a Scala *self type* [21] to declare that it must be mixed in with implementations of the `ControlFlow` and `Variables` interfaces.

```
trait LivenessImpl extends Liveness {
    self : Liveness with Variables with ControlFlow =>
```

```
import kiama.attribution.DynamicAttribution._

case class Foreach (cond : Var, body : Stm) extends Stm

object DataflowForeach {
    Dataflow.succ += {
        case t @ Foreach (_, body) => following (t) + body }
    }

    Dataflow.following +=
        childAttr {
            _ => {
                case t @ Foreach(_, body) => following (t) + body
            }
        }
    }
}
```

Fig. 7. Dynamic attribute grammar extension.

```
    ...
    ... definitions of in and out as before
    ...
}
```

Finally, an implementation of the dataflow solution can be formed by mixing together implementations of the three modules.

```
object Dataflow extends LivenessImpl with VariablesImpl
    with ControlFlowImpl
```

This approach allows modules to be composed with alternative implementations without being changed or even recompiled because the types ensure that the composition is valid.

4.2 *Dynamically Extensible Attribute Definitions*

Kiama uses functions to implement attributes, represented by `CachedAttribute` and other types. In this subsection, we illustrate the flexibility of this approach by adding a new dynamic form of attributes. Kiama's `DynamicAttribution` module defines attributes using the interfaces shown earlier and adds the `+=` operator to enable an attribute definition to be dynamically extended. Therefore, attribute grammar specifications can be separately compiled, dynamically loaded into the Java Virtual Machine, and added to an existing definition. This makes it possible to distribute language extensions in the form of binary plugins.

The extension operator is illustrated by Figure 7 that extends the dataflow example of Section 2.2 by adding a `Foreach` construct. The body of `DataFlowForeach` is a set of statements; the extension is only activated if these are executed. Each invocation of `+=` on an attribute adds a new definition to an internally maintained list of partial functions for the attribute. Inspired by the Disposable pattern [17], we introduce a method similar to the `using` statement in languages such as C#. With this technique, we can activate the extension as follows:

```
using (DataflowForeach) {
  ... // evaluate attributes using the extension
}
```

The extension is only active in the scope of the block of code, and any defini-

tions added are removed after it completes. The `using` method is implemented as follows: [3]

```
def using[T] (attributeInitializer : => AnyRef) (block : => T) =
    try {
        use (attributeInitializer)
        block
    } finally {
        endUse (attributeInitializer)
    }
```

That is, it uses two helper methods to first activate and keep track of new definitions, then evaluates the block, and finally removes the definitions again. Therefore, `using` allows extensions to be combined easily in a disciplined, scoped fashion.

5 Evaluation

We evaluate the performance of attribute evaluation in Kiama by a comparison to a handwritten Scala implementation and to a generated Java attribute evaluator. For the former, we specify attributes as regular methods in the AST classes, and perform caching by hand, at a cost of modularity and boilerplate code. For the latter, we compare to JastAdd, which, like Kiama, uses the Java platform and supports reference and circular attributes [11]. JastAdd has been successfully used to implement a full-featured Java 1.5 compiler that offers performance that can compete with handwritten implementations [9].

As a test case, we use the JastAdd example PicoJava specification from [27], which has 18 abstract syntax productions and 10 attributes to perform name and type analysis. We tested evaluation performance for relatively large, generated input programs. Since PicoJava only supports class definitions and not methods, our input classes contain 150 nested class definitions.

Figure 8 shows our benchmark results. The LOC column shows the number of lines of non-commented code to implement the AST and attribute grammar in each specification. The timings show the amount of time used for 100 runs evaluating the `errors` attribute that uses the other attributes to check for naming problems and cycles in the inheritance hierarchies. We first constructed a list of 100 inputs and evaluated the attribute for each input. In a second series of tests, we constructed only a single input in each run, ensuring that older inputs could be garbage collected, minimizing memory overhead of the benchmark. We used this approach because the input classes are particularly large for JastAdd which uses fields for caching. In both cases, we only timed the attribute evaluation process, ignoring input/output and tree creation overhead

The original JastAdd specification only used caching on selected attributes, which for our test cases appeared to lead to a decrease in performance. Thus, we created a variation where all attributes were cached, and finally a further variation that disabled JastAdd's use of rewrite rules. The Kiama implementation is a direct translation of this last variant. The results indicate that Kiama, while not

[3] A parameter type `=> T` denotes a call-by-name parameter.

	LOC	*List of classes*	*Single classes*
Java: JastAdd	252	4010 ms	3459 ms
Java: JastAdd (full caching)	252	1767 ms	952 ms
Java: JastAdd (full caching, no rewrites)	243	1260 ms	860 ms
Scala: Kiama (full caching, no rewrites)	262	1889 ms	2435 ms
Scala: Handwritten	424	862 ms	543 ms

Fig. 8. Lines of non-commented code (LOC) in the benchmark specifications and times to evaluate the `errors` attribute of a large PicoJava input program.

using code generation and having had little performance optimisation, provides competitive performance to JastAdd and adds only limited overhead to a handwritten specification. Of course, like JastAdd, Kiama offers superior modularity and a more concise notation than is possible with the handwritten implementation.

6 Discussion and Related Work

This section briefly compares the approach taken to develop the Kiama attribution library with generator-based systems that feature a dynamic evaluation approach. We are not aware of another attribute grammar system that uses pure embedding.

In many ways, Kiama has been inspired by the JastAdd [11] system and the features provided are similar. JastAdd provides an object-oriented variation of attribute grammars, supporting inheritance in their definition and references as attribute values [11]. Like JastAdd, Kiama is based on the Java platform, but makes use of the Scala language rather than a pre-processor approach.

The JastAdd approach to attribute evaluation might be characterised as "roll your own" laziness for Java. Scala does have lazy values, but to use them in Kiama would require attribute definitions to reside in the abstract syntax classes, which goes against modularity. Therefore we use the same general approach as JastAdd, but cache the values in attribute objects rather than in the tree nodes. This design implies some space overhead but we haven't observed it to be a problem in practice.

A number of systems use lazy functional languages to define evaluators as circular programs [13]. The most prominent recent projects are LRC [23], Silver [25], the UU AG system [2] and first-class attribute grammars [5]. Built-in laziness means that explicit scheduling of attributes is avoided. Fully circular attributes are not possible by default but a form of circularity can be obtained [1]. In contrast, Kiama's approach is more work to implement but more flexible because we have lightweight, fine-grained control over the mechanisms used to evaluate attributes while retaining the property that schedules are computed implicitly.

All of the systems cited use a special-purpose front-end or pre-processor to translate their attribute grammars into an implementation language, Java in the case of JastAdd and Haskell for the other systems. Since attribute equations in these systems are largely written in the syntax of the target language, a fairly high level of integration is achieved. Kiama removes the generator completely. While a custom input language is often desirable, the benefits can be outweighed by the simplicity and light-weight nature of an approach that doesn't need a generator with its asso-

ciated learning curve and influence on the build process. Scala's expressive nature limits the sacrifices that must be made when making this tradeoff.

The first-class attribute grammars work [5] gets the closest to a pure embedding since attributes are first-class citizens that can be combined using combinator functions. As such, it shows similarities to our dynamically extensible attributes. However, the syntax used is supported by a pre-processor, rather than using pure Haskell. Based on the Haskell type checker, first-class attribute grammars prevent errors where the use of an attribute does not match its type. Errors due to cyclic dependencies or a mismatch between attribute equations and grammar productions are not reported. In earlier work, De Moor et al [6] used a Rémy-style record calculus to detect errors of the latter category, but this was found to be too restrictive.

One advantage of a generator-based approach is the ability to check the attribute grammar for correctness at generation time. For example, completeness and well-formedness checks [7] give confidence that the generated evaluator is not incomplete. In Kiama, precise checking of this kind is not always possible, particularly if syntax extensibility is desired. A Scala case class can be marked `sealed` which means that it cannot be extended outside the current module. When compiling a pattern match against a sealed class, the Scala compiler can emit warnings if the patterns are not complete, giving Kiama a form of completeness checking.

Kiama's encoding of the abstract syntax grammar in case classes also removes the possibility of some grammar-based checks. For example, in the Repmin example of Section 2.1 a run-time type check was necessary to ensure that the parent of a tree node was also a tree node. This check would not be necessary in a grammar-based generator, since the relationships between non-terminals could be determined statically. In practice, however, checks of this kind are not needed often and therefore do not outweigh the advantages of using standard Scala case classes as Kiama's tree representation.

As mentioned in Section 4, many attribute grammar systems allow the grammar to be written as separate "aspects" that are automatically "woven" together at generation time. In contrast, Kiama requires explicit descriptions of composition in the specification. While automatic composition of aspects is certainly convenient, in a general-purpose language setting where explicit composition is the norm its absence is not really felt.

7 Conclusion and Future Work

Dynamically-scheduled attribute grammars are a powerful language processing paradigm that has been the focus of many generator-based implementations. In most cases, a general purpose language is used to express attribute computations. The Kiama attribution library removes the generation step by using Scala to write the whole attribute grammar. The resulting system is lightweight and easy to understand, yet capable of competing in expressivity and performance with JastAdd, a mature generator-based system which uses a similar evaluation method.

The Scala features used by Kiama are present in one form or another in other

languages, although usually not together. Scala's powerful expression language and first-class functions are well complemented by object-oriented features for state encapsulation and modularity. More than any other feature, Kiama benefits most from Scala's pattern-matching anonymous functions that support the clean and natural attribute equation notation.

Kiama is in active development. For example, we are adding collection attributes [4,15]. Scala's ability to extend traits that define values (as opposed to methods) is being improved, which we plan to use to provide better support for defining a single attribute in multiple modules. The general question of analysis for embedded languages is also interesting for Kiama since it could lead to a solution that is both modular and provides better static completeness guarantees.

References

[1] Augusteijn, A., "Functional programming, program transformations and compiler construction," Ph.D. thesis, Department of Computing Science, Eindhoven University of Technology, The Netherlands (1993).

[2] Baars, A., D. Swierstra and A. Löh, *UU AG System User Manual*, Department of Computer Science, Utrecht University, September (2003).

[3] Bird, R., *Using circular programs to eliminate multiple traversals of data*, Acta Informatica **21** (1984), pp. 239–250.

[4] Boyland, J. T., *Remote attribute grammars*, J. ACM **52** (2005), pp. 627–687.

[5] de Moor, O., K. Backhouse and S. Swierstra, *First-class attribute grammars*, Informatica **24** (2000), pp. 329–341.

[6] de Moor, O., S. Peyton-Jones and E. Van Wyk, *Aspect-oriented compilers*, in: *Proceedings of International Symposium on Generative and Component-based Software Engineering*, LNCS **1799** (1999), pp. 121–133.

[7] Deransart, P., M. Jourdan and B. Lorho, "Attribute Grammars: Definitions, Systems and Bibliography," Lecture Notes in Computer Science **323**, Springer-Verlag, Berlin, Germany, 1988.

[8] Ekman, T. and G. Hedin, *Modular name analysis for Java using JastAdd*, in: *International Summer School in Generative and Transformational Techniques in Software Engineering*, Lecture Notes in Computer Science **4143** (2006), pp. 422–436.

[9] Ekman, T. and G. Hedin, *The JastAdd extensible Java compiler*, in: *Proceedings of the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA'07)* (2007), pp. 1–18.

[10] Hedin, G., *Reference Attributed Grammars*, Informatica (Slovenia) **24** (2000), pp. 301–317.

[11] Hedin, G. and E. Magnusson, *JastAdd: an aspect-oriented compiler construction system*, Sci. Comput. Program. **47** (2003), pp. 37–58.

[12] Hudak, P., *Modular domain specific languages and tools*, in: *Proceedings of the 5th International Conference on Software Reuse* (1998), pp. 134–142.

[13] Johnsson, T., *Attribute grammars as a functional programming paradigm*, in: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (1987), pp. 154–173.

[14] Jourdan, M., *An optimal-time recursive evaluator for attribute grammars*, in: *Proceedings of the International Symposium on Programming*, Springer, 1984, pp. 167–178.

[15] Magnusson, E., T. Ekman and G. Hedin, *Extending attribute grammars with collection attributes - evaluation and applications*, in: *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation* (2007).

[16] Magnusson, E. and G. Hedin, *Circular Reference Attributed Grammars-their Evaluation and Applications*, Electronic Notes in Theoretical Computer Science **82** (2003), pp. 532–554.

[17] Mariani, R., *Garbage collector basics and performance hints*, MSDN Library. http://msdn.microsoft.com/en-us/library/ms973837.aspx (2003).

[18] Mernik, M., J. Heering and A. M. Sloane, *When and how to develop domain-specific languages*, Computing Surveys **37** (2005), pp. 316–344.

[19] Nilsson-Nyman, E., G. Hedin, E. Magnusson and T. Ekman, *Declarative intraprocedural flow analysis of Java source code*, in: *Proceedings of the 8th Workshop on Language Descriptions, Tool and Applications*, 2008.

[20] Odersky, M., L. Spoon and B. Venners, "Programming in Scala," Artima Press, 2008.

[21] Odersky, M. and M. Zenger, *Scalable component abstractions*, in: *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages and Applications*, 2005, pp. 41–57.

[22] Paakki, J., *Attribute grammar paradigms—a high-level methodology in language implementation*, ACM Computing Surveys **27** (1995), pp. 196–255.

[23] Saraiva, J., "Purely Functional Implementation of Attribute Grammars," Ph.D. thesis, Department of Computer Science, Utrecht University, The Netherlands (1999).

[24] Sloane, A. M., *Experiences with domain-specific language embedding in Scala*, in: *Proceedings of the 2nd International Workshop on Domain-Specific Program Development*, 2008.

[25] Van Wyk, E., D. Bodin, J. Gao and L. Krishnan, *Silver: an Extensible Attribute Grammar System*, Electronic Notes in Theoretical Computer Science (ENTCS) **203** (2008), pp. 103–116.

[26] Vogt, H. H., S. D. Swierstra and M. F. Kuiper, *Higher order attribute grammars*, in: *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation* (1989), pp. 131–145.

[27] *Picojava checker*, http://jastadd.cs.lth.se/examples/PicoJava/.

Syntactic Language Extension via an Algebra of Languages and Transformations

Jacob Andersen [1]

Department of Computer Science
Aarhus University; Aarhus, Denmark

Claus Brabrand [2]

IT University of Copenhagen
Copenhagen, Denmark

Abstract

We propose an algebra of languages and transformations as a means for extending languages syntactically. The algebra provides a layer of high-level abstractions built on top of *languages* (captured by context-free grammars) and *transformations* (captured by constructive catamorphisms).

The algebra is *self-contained* in that any term of the algebra specifying a transformation can be *reduced* to a catamorphism, before the transformation is run. Thus, the algebra comes "for free" without sacrificing the strong safety and efficiency properties of constructive catamorphisms.

The entire algebra as presented in the paper is implemented as the Banana Algebra Tool which may be used to syntactically extend languages in an incremental and modular fashion via algebraic composition of previously defined languages and transformations. We demonstrate and evaluate the tool via several kinds of extensions.

Key words: Languages; transformation; syntactic extension; macros; context-free grammars; catamorphisms; bananas; algebra.

1 Introduction and Motivation

We propose an algebra of 16 operators on languages and transformations as a *simple*, *incremental*, and *modular* way of specifying *safe* and *efficient* syntactic language extensions through algebraic composition of previously defined languages and transformations.

[1] Email: jacand@cs.au.dk
[2] Email: brabrand@itu.dk

Extension is *simple* because we base ourselves on a well-proven and easy-to-use formalism for well-typed syntax-directed transformations known as *constructive catamorphisms*. These transformations are specified relative to a source and a target language which are defined via context-free grammars (CFGs). Catamorphisms have previously been studied and proven sufficiently expressive as a means for extending a large variety of programming languages via transformation [5,6,7]. Hence, the main focus of this paper lies not so much in addressing the expressiveness and which transformations can be achieved as on showing how algebraic combination of languages and transformations results in highly modular and incremental language extension. *Incremental* and *modular* means that any previously defined languages or transformations may be composed algebraically to form new languages and transformations. *Safety* means that the tool statically guarantees that the transformations always terminate and only map syntactically legal input terms into syntactically legal output terms; *Efficiency* means that any transformation is guaranteed to run in linear time (in the size of input and generated output).

An important property of the algebra which is built on top of catamorphisms is that it is "self-contained" in the sense that any term of the algebra may be *reduced* to a constant catamorphism, at compile-time. This means that all high-level constructions offered by the algebra (including composition of languages and transformations) may be dealt with at compile-time, before the transformations are run, without sacrificing the strong safety and efficiency guarantees.

Everything presented in the paper has been implemented in the form of The Banana Algebra Tool which, as argument, takes a transformation term of the algebra which is then analyzed for safety and *reduced* to a constant catamorphism which may subsequently be run to transform an input program.

The tool may be used for many different transformation purposes, such as transformation between different languages (e.g., for translating Java programs into HTML documentation in the style of JavaDoc or for prototyping lightweight domain-specific language compilers), transforming a given language (e.g., the CPS transformation), format conversion (e.g., converting BibTex to BibTeXML). However, in this paper we will focus on *language extension* for which we have the following usage scenarios in mind: 1) Programmers may extend existing languages with their own macros; 2) Developers may embed domain-specific languages (DSLs) in host languages; 3) Compiler writers may implement only a small core and specify the rest externally; and 4) Developers or teachers may define languages incrementally by stacking abstractions on top of each other. We will substantiate these usage claims in Section 6.

The approach captures the niche where full-scale compiler generators as outlined in Section 7 are too cumbersome and where simpler techniques for syntactic transformation are not expressive or safe enough, or do not have sufficient support for incremental development.

Our contributions include the design of an algebra of languages and trans-

formations for incremental and modular syntactic language extension built on top of catamorphisms; a proof-of-concept tool and implementation capable of working with concrete syntax; and an evaluation of the algebraic approach.

2 Catamorphisms

A *catamorphism* (aka., *banana* [16]) is a generalization of the *list folding* higher-order function known from functional programming languages which processes a list and builds up a return value. However, instead of working on lists, it works on any inductively defined datatype. Catamorphisms have a strong category theoretical foundation [16] which we will not explore in this paper. A catamorphism associates with each constructor of the datatype a *replacement evaluation function* which is used in a transformation. Given an input term of the datatype, a catamorphism then performs a recursive descent on the input structure, effectively deconstructing it, and applies the replacement evaluation functions in a bottom-up fashion recombining intermediate results to obtain the final output result.

Many computations may be expressed as catamorphisms. As an example, let us consider an inductively defined datatype, `list`, defining non-empty lists of numbers:

$$\texttt{list} \;=\; \texttt{Num}\ \mathbb{N} \;\mid\; \texttt{Cons}\ \mathbb{N}\ \texttt{*}\ \texttt{list}$$

The sum of the values in a list of numbers may easily be defined by a catamorphism, by replacing the `Num`-constructor by the identity function on numbers ($\lambda n.n$) and the `Cons`-constructor by addition on numbers ($\lambda(n,l).n{+}l$), corresponding to the following recursive definition:

$$\begin{array}{rcl} [\![\texttt{Num}\ n]\!] & = & n \\ [\![\texttt{Cons}\ n\ l]\!] & = & n{+}[\![l]\!] \end{array}$$

One of the main advantages of catamorphisms is that recursion over the structure of the input is completely separated from the construction of the output. In fact, the recursion is completely determined from the input datatype and is for that reason often only specified implicitly. Since the sum catamorphism above maps terms of type `list` to natural numbers $\mathbb{N}$, it may be uniquely identified with its replacement evaluation functions; in this case with a replacement evaluation function for the `Num`-constructor of type $\mathbb{N} \to \mathbb{N}$ and a replacement function of type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ for `Cons`). Catamorphisms are often written in the so-called banana brackets "$(\!|\cdots|\!)$" [16]:

$$(\!|\ \lambda n.n\ ,\ \lambda(n,l).n{+}l\ |\!)$$

2.1 Constructive Catamorphisms

Constructive catamorphisms are a restricted form of catamorphisms where only output-typed *reconstructors* are permitted as replacement evaluator functions. Reconstructors are just constructor terms from (possibly different) inductively defined datatypes wherein the arguments to the constructive catamorphism may be used. For instance, we can transform the lists into binary trees of the `tree` datatype:

$$\texttt{tree} \;=\; \texttt{Nil} \;\mid\; \texttt{Leaf}\ \mathbb{N} \;\mid\; \texttt{Node}\ \mathbb{N}\ \texttt{*}\ \texttt{tree}\ \texttt{*}\ \texttt{tree}$$

using a constructive catamorphism:

$$\begin{array}{rcl} [\![\texttt{Num}\ n]\!] & = & \texttt{Leaf}\ n \\ [\![\texttt{Cons}\ n\ l]\!] & = & \texttt{Node}\ n\ \texttt{(Nil)}\ [\![l]\!] \end{array}$$

Although very simple, capable of trivial recursion only, we claim that this kind of constructive catamorphisms provide a basis for programming language extension. We shall investigate this claim in the following section.

2.2 Safety and Efficiency

Constructive catamorphisms have a lot of interesting properties; they can be statically verified for *syntactic safety*, are guaranteed to terminate, and to run in linear time.

A constructive catamorphism, c, is *typed* with a source language, l_s, and a target language, l_t, as in "$l_s \rightarrow l_t$". The languages can be given either as a *datatype* (at the abstract syntactic level) as above, or as a *CFG* (at the concrete syntactic level). A constructive catamorphism is said to be *syntactically safe* if it only produces syntactically valid output terms, $\omega_t \in \mathcal{L}(l_t)$, given syntactically valid input terms, $\omega_s \in \mathcal{L}(l_s)$:

$$\forall \omega \in \mathcal{L}(l_s) \Rightarrow c(\omega) \in \mathcal{L}(l_t)$$

In addition to a *language typing* ($l_s \rightarrow l_t$), we also need a *nonterminal typing*, τ, which for each of the nonterminals of the input language specifies onto which nonterminal of target language they are mapped.

If we name the source and target languages of the above example `Lists` and `Trees` respectively, the language typing then becomes "`Lists -> Trees`" and the nonterminal typing, τ, is "`[list -> tree]`". (The reason for the angled bracket convention is that there may be multiple nonterminals in play, in which case multiple mappings are written as a comma separated list inside the brackets.)

In order to verify that a catamorphism, $(\!|\, l_s \rightarrow l_t\ [\tau]\ c \,|\!)$ is syntactically safe, one simply needs to check that each of the catamorphism's reconstructor terms (e.g., "`Node` n `(Nil)` $[\![l]\!]$") are valid syntax, assuming that each of its argument usages (e.g., $[\![l]\!]$) are valid syntax of the appropriate type (in this

case l has source type `list` which means that $[\![l]\!]$ has type $\tau(\texttt{list}) = \texttt{tree}$). We refer to [1] for a formal treatment of how to verify syntactic safety.

Constructive catamorphisms are highly efficient. Asymptotically, they run in linear time in the size of the input and output: $O(|\omega| + |c(\omega)|)$.

3 Language Extension

We will now illustrate—using deliberately simple examples—how constructive catamorphisms may be used to extend programming languages and motivate the idea of programming language extension. To this end, let us consider the core λ-Calculus (untyped, without constants) whose syntactic structure may be defined by the following datatype:

```
exp  =  Var id  |  Lam id * exp  |  App exp * exp
```

In the following, we will investigate how to extend the λ-Calculus using catamorphisms; in particular, we will look at two well-known extensions, namely that of numerals and booleans.

3.1 Extension: Numerals

A common extension of the core λ-Calculus is that of *numerals*; the calculus is extended with a construction representing *zero*, and unary constructors representing the *successor* and *predecessor* of a numeral. These constructions may be combined to represent any natural numbers in unary encoding and for performing numeric calculations. The syntax of the calculus is then extended to the language, `LN`:

```
exp  =  Var id  |  Lam id * exp  |  App exp * exp  |
        Zero  |  Succ exp  |  Pred exp
```

We will now show how a catamorphism may be used to transform the extended language, `LN`, into the core λ-Calculus, `L`, using a basic encoding of numerals which represents *zero* as the identity function ($\lambda z.z$), and a number n as follows:

$$\overbrace{\lambda\, s\, .\, \lambda\, s\, .\, \cdots\, \lambda\, s\, .}^{n \text{ lambdas}}\ \overbrace{\lambda\, z\, .\, z}^{zero}$$

There are many other possible encodings of numerals, including the more commonly used Church numeral representation, but the choice of encoding is not of primary interest here, so we will just use the simpler alternative to illustrate the point. We can now extend the λ-Calculus with numerals as a constructive catamorphism of type "`LN -> L [exp -> exp]`":

$$
\begin{array}{rcl}
[\![\texttt{Var}\ V]\!] & = & \texttt{Var}\ [\![V]\!] \\
[\![\texttt{Lam}\ V\ E]\!] & = & \texttt{Lam}\ [\![V]\!]\ [\![E]\!] \\
[\![\texttt{App}\ E_1\ E_2]\!] & = & \texttt{App}\ [\![E_1]\!]\ [\![E_2]\!] \\
[\![\texttt{Zero}]\!] & = & \texttt{Lam z (Var z)} \\
[\![\texttt{Succ}\ E]\!] & = & \texttt{Lam s}\ [\![E]\!] \\
[\![\texttt{Pred}\ E]\!] & = & \texttt{App}\ [\![E]\!]\ \texttt{(Lam z (Var z))}
\end{array}
$$

The first three rules just trivially recurse through the input structure producing an identical output structure. Zero becomes the identity function, successor adds a "lambda s" in front of the encoding of the argument, and predecessor peels off one lambda by applying it to the identity function (note that the predecessor of zero is thus consequently defined as zero). This will, for instance, map `Succ Zero` to its encoding `Lam s (Lam z (Var z))`.

3.2 Other Extensions

Similarly, the core λ-Calculus may easily be extended with *booleans* (via nullary constructors `True` and `False`, and a ternary `If`) yielding a syntactically extended language `LB` which could then be transformed to the core λ-calculus by a constructive catamorphism with typing "`LB -> L [exp -> exp]`":

$$
\begin{array}{rcl}
[\![\texttt{True}]\!] & = & \texttt{Lam a (Lam b (Var a))} \\
[\![\texttt{False}]\!] & = & \texttt{Lam a (Lam b (Var b))} \\
[\![\texttt{If}\ E_1\ E_2\ E_3]\!] & = & \texttt{App (App}\ [\![E_1]\!]\ [\![E_2]\!]\texttt{)}\ [\![E_3]\!]
\end{array}
$$

Note that we have omitted the three lines of "identity transformations" for variables, lambda abstraction, and application.

Along similar lines, the λ-Calculus could be further extended with *addition*, *multiplication*, *negation*, *conjunction*, *lists*, *pairs*, and so on, eventually converging on a full-scale programming language. To substantiate the claim that this forms an adequate basis for language extension, we have extended the λ-Calculus towards a language previously used in teaching functional languages; "Fun" (cf. Section 6).

4 Algebra of Languages and Transformations

Investigating previous work on syntactic macros and transformations [5,6,7] has revealed an interesting and recurring phenomenon in that macro extensions follow a certain pattern. The first hint in this direction is the effort involved in the first three lines of the constructive catamorphisms which are there merely to specify the "identity transformation" on the core λ-Calculus. That effort could be alleviated via explicit language support.

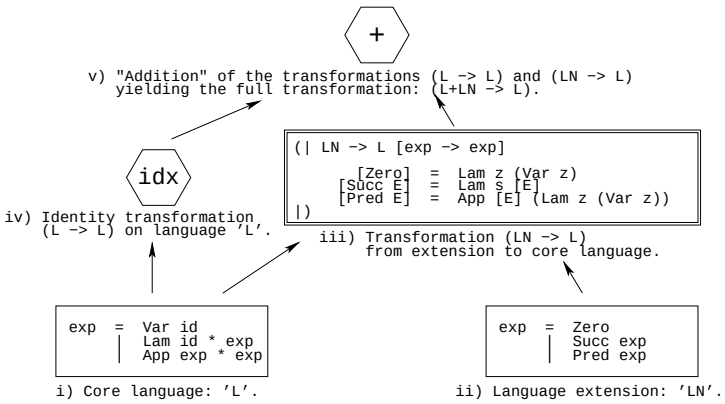


Fig. 1. Common pattern in language extension (here extending the λ-Calculus with numerals.)

In fact, every such language extension can be broken into the same five ingredients (some of which are *languages*, some of which are *transformations*), depicted in Figure 1: i) a core language that is to be extended (e.g., the λ-Calculus); ii) a language extension of that language [3] (e.g., the extension with

[3] Note that we refer to the extended language as excluding the core language.

numerals); iii) an identity transformation on the core language; iv) a transformation that maps the extended language to the core language; and v) a notion of "addition" of the identity transformation and the small transformation of the language extension to the core language.

4.1 The Algebra

The five ingredients above can be directly captured by five algebraic operators. First, cases i) and ii) correspond to a *constant language* operator which may be modeled by a context-free grammar (with "named productions" for attaching transformations). Second, case iii) corresponds to a constant transformation which may be given as an output-typed constructive catamorphism, c, typed with the source and target languages of the transformation (and a nonterminal typing, τ). Third, case iv) corresponds to an operator taking a language l and turning it into the identity transformation ($l \rightarrow l$) on that language. Fourth, a notion of addition on transformations, taking two transformations $l_s \rightarrow l_t$ and $l'_s \rightarrow l'_t$ yielding a transformation: $(l_s \oplus_l l'_s) \rightarrow (l_t \oplus_l l'_t)$ where "$\oplus_l$" is addition on languages. Language addition is defined as the *union* of the individual productions (transformation addition as the union of the catamorphic reconstructors), which in both cases ensure that addition is *idempotent*, *symmetric*, *associative*, and *commutative*. For a formal definition of addition on languages and transformations, we refer to [1].

Note that with these operations, it is very easy to obtain a transformation combining *both* the extension of numerals and booleans; simply "add" the two transformations.

Although the above algebraic operations are enough to make all the extensions of the previous chapter, we would like to motivate a couple more algebraic operators on languages and transformations. Note that even though the design, and choice of operators arose through an iterative process, we have tried to divide and categorize the motivations for the constructions into two categories; operators accommodating respectively *modular* and *incremental* language extension. The complete syntax for the algebra is presented in Figure 2. (The rules for *language constants*, *transformation constants*, *language addition*, *transformation addition*, and *identity transformations* are numbered `L1`, `X1`, `L4`, `X4`, and `X6`, respectively.) Of course, it is possible to add even more operators to the algebra; however, the ones we have turn out to be sufficient to conveniently extend the λ-Calculus incrementally all the way to the Fun programming language. These ideas are pursued in the remainder of the paper which also includes an evaluation of the whole algebraic approach. For a formal specification of the semantics of the algebra, see the Appendix (for a specification of the underlying languages and transformations, see [1]).

L	$\rightarrow_{\texttt{L1}}$	l	X	$\rightarrow_{\texttt{X1}}$	$(\!\vert L \rightarrow L\ [\tau]\ c \vert\!)$
	$\rightarrow_{\texttt{L2}}$	v		$\rightarrow_{\texttt{X2}}$	w
	$\rightarrow_{\texttt{L3}}$	L `\` L		$\rightarrow_{\texttt{X3}}$	X `\` L
	$\rightarrow_{\texttt{L4}}$	L `+` L		$\rightarrow_{\texttt{X4}}$	X `+` X
	$\rightarrow_{\texttt{L5}}$	`src (` X `)`		$\rightarrow_{\texttt{X5}}$	X $\circ$ X
	$\rightarrow_{\texttt{L6}}$	`tgt (` X `)`		$\rightarrow_{\texttt{X6}}$	`idx (` L `)`
	$\rightarrow_{\texttt{L7}}$	`let` v`=`L `in` L		$\rightarrow_{\texttt{X7}}$	`let` v`=`L `in` X
	$\rightarrow_{\texttt{L8}}$	`letx` w`=`X `in` L		$\rightarrow_{\texttt{X8}}$	`letx` w`=`X `in` X

(a) Algebra of languages (L)... (b) ...and transformations (X).

Fig. 2. Syntax of the algebra.

4.2 Modular language extension

In order to permit modular language development and separate each of the ingredients in a transformation, we added *local definition* mechanism via the standard `let-in` functional programming local binder construction. Thus, we add to the syntax of both languages and transformations; *variables* (Figure 2, rules `L2` and `X2`) and *local definitions* (Figure 2, rules `L7`, and `X7`).

In practice, it turns out to be useful to also be able to define (local) *transformations* while specifying *languages*; and, orthogonally, to define (local) *languages* while specifying *transformations*. Hence, we add the local definitions `L8` and `X8` to Figure 2.

4.3 Incremental language extension

Transformations are frequently specified incrementally in terms of previously defined languages and transformations. To accommodate such use we added a means for designating the *source* and *target* languages of a transformation along with a means for *restricting* a language and a transformation (i.e., restricting the source language of a transformation). By restriction, we take "L_1 `\` L_2" to yield a language identical to L_1, but where all productions also mentioned by name in L_2 have been eliminated. (The operators mentioned are listed as rules `L5`, `L6`, `L3`, and `X3` of Figure 2.)

Also, transformations are frequently expressed via intermediate syntactic constructions for either simplicity or legibility. For instance, notice how two of the catamorphic reconstructors in the transformation of Section 3.1 both use the identity lambda abstraction `Lam z (Var z)`. Here, one could specify this transformation incrementally, by using an intermediary language, `LI`, enriched with identity as an explicit nullary construction:

```
exp  =  Var id  |  Lam id * exp  |  App exp * exp  |  Id
```

Although on such a small example, there is little to gain in terms of simplicity and/or legibility, it illustrates the general principle of incremental language

extension. The transformation ("`LN -> L`") can now be simplified to "`ln2li: LN -> LI`":

$$\begin{array}{rcl} [\![\texttt{Zero}]\!] & = & \texttt{Id} \\ [\![\texttt{Succ } E]\!] & = & \texttt{Lam s } [\![E]\!] \\ [\![\texttt{Pred } E]\!] & = & \texttt{App } [\![E]\!] \texttt{ (Id)} \end{array}$$

Which is subsequently composed with the tiny transformation that desugares the identity-enriched language to the core λ-Calculus, "`li2l: LI -> L`":

$$[\![\texttt{Id}]\!] = \texttt{Lam z (Var z)}$$

Not surprisingly, when we do this experiment using the tool, the transformation "`li2l` $\circ$ `li2ln`" produces the exact same transformation as the directly specified constant transformation in Section 3.1. To enable such incremental development, we added *composition* as an operator on transformations (cf. Figure 2, rule `X5`).

Note that none of the operators go beyond the expressivity of constructive catamorphisms in that any language term can be statically reduced to a context-free grammar; and any transformation term to a catamorphism.

An important advantage of an algebraic approach is that several algebraic laws hold which give rise to simplifications (e.g., "L `+` $L \equiv L$", "L_1 `+` $L_2 \equiv L_2$ `+` L_1", "L_1 `+ (`L_2 `+` L_3`)` $\equiv$ `(`L_1 `+ (`L_2`) +` L_3", "`src(id(`L`))` $\equiv L$") to mention but a few. (For a formal specification of the reductin and semantics of the operators, see the Appendix.)

5 Tool and Implementation

In order to validate the algebraic approach, we have implemented everything in the form of The Banana Algebra Tool which we have used to experiment with different forms of language extensions.

5.1 Abstract vs. Concrete Syntax

A key issue in building the tool was the choice of whether to work with *abstract* or *concrete* syntax. Everything we have presented so far has been working exclusively on the abstract syntactic level. For practical usability of the tool, however, it turns out to be more convenient to work on the concrete syntax. Note that because of the addition operators of the algebra, it is important that particular choice of parsing algorithm be closed under union.

Figure 3 illustrates the difference between using abstract and concrete syntax for specifying transformations. Figure 3(a) depicts a fragment of a grammar for a subset of Java that deals with associativity and precedence of expressions by factorizing operators into several distinct levels according

```
Exp.or    : Exp1 "||" Exp  ;
   .exp1  : Exp1           ;
Exp1.and  : Exp2 "&&" Exp1 ;
    .exp2 : Exp2           ;
Exp2.add  : Exp3 "+" Exp2  ;
    .exp3 : Exp3           ;
...
Exp7.neg  : "!" Exp8       ;
    .exp8 : Exp8           ;
Exp8.par  : "(" Exp ")"    ;
    .var  : Id             ;
    .num  : IntConst       ;
```

(a) Java grammar fragment.

```
Stm.repeat =
  Stm.do(<1>,
    Exp.exp1(
      Exp1.exp2(
        Exp2.exp3(
          Exp3.exp4(
            Exp4.exp5(
              Exp5.exp6(
                Exp6.exp7(
                  Exp7.neg(
                    Exp8.par(<2>)
                  ))))))))) ;
```

(b) *Abstract* syntax.

```
Stm.repeat =
  'do <1> while (!(<2>));' ;
```

(c) *Concrete* syntax.

Fig. 3. Example specifying transformations using *abstract* vs. *concrete* syntax. (For emphasis, we have underlined the negation and parenthesis constructions.)

to operator precedence (as commonly found in programming language grammars); in this case, there are nine levels from `Exp` and `Exp1` all the way to `Exp8`.

Now suppose we were to *extend* the syntax of Java by adding a new statement, `repeat-until`, with syntax: `"repeat" Stm "until" "(" Exp ")" ";"`. Such a construction can easily be transformed into core Java by desugaring it into a `do-while` with a negated condition. Figure 3(b) shows how this would be done at the abstract syntactic level, using abstract syntax trees (ASTs). Transformation arguments are written in angled brackets; e.g., `<1>` and `<2>` (as explained later). Since negation is found at the eighth precedence level (in `Exp7`), the AST fragment for specifying the negated conditional expression would have to take us from `Exp` all the way to `Exp7`, add the negation "`Exp7.neg(...)`", before adding the parentheses "`Exp8.par(...)`" and the second argument, "`<2>`" (which contains the original expression that was to be negated). Figure 3(c) specifies the same transformation, but at the concrete syntactic level, using strings instead of ASTs. At this level, there is no need for dealing explicitly with such low-level considerations which are more appropriately dealt with by the parser.

Interestingly, if the grammar of a language is unambiguous and we choose a canonical unparsing, we may move reversibly between abstract syntax trees and concrete syntactic program strings. Since we have such a recent ambiguity analysis [3], we have chosen to base the tool on concrete syntax. However, transformations may also be written in abstract syntax as in Figure 3(b).

5.2 Underlying technologies

Figure 4 depicts the transformation process. The Banana Algebra Tool is currently based on XSugar [5] and XSLT [4], but the tool is easily modified to use other underlying tools (only code generation is affected by these choices). We use XSugar for parsing a concrete term of the source language (e.g., "`succ`

[4] http://www.w3.org/

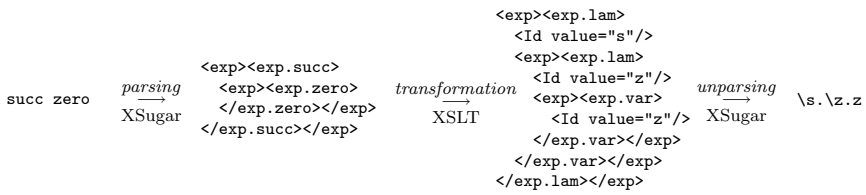


Fig. 4. The transformation process.

`zero`") to an AST represented in XML. (XSugar uses an eager variant of Earley's algorithm, capable of parsing any CFG, and a conservative ambiguity analysis [3] which may be used to verify unambiguity of all languages involved.) Then, we use XSLT for performing the catamorphic transformation from source AST to target AST. Finally, XSugar unparses the AST into an output term of the target language.

5.3 Other implementation issues

We found it convenient to permit lexical structure to be specified using regular expressions, as often encountered in parser/scanner tools. However, the tool currently considers this an atomic terminal layer that cannot be transformed.

We handle whitespace via permitting a special whitespace terminal named "`$`" to be defined (it defaults to the empty regular expression). The semantics is that the whitespace is interspersed *between* all terminal and nonterminals on the right-hand-side of all productions. For embedded languages, it might be interesting to have finer grained control over this, but that is currently not supported by our tool.

In the future, it would be interesting to also add a means for *alpha conversion* and *static semantics* checks on top of the syntactic specifications

6 Examples and Evaluation

The tool can be used for any syntax-directed transformation that can be expressed as catamorphisms (which includes all the transformations of Metafront [7] and XSugar [5]). This includes translation between different languages, transformations on a language, and format conversion, but here we will focus on *language extension* from each of the "four scenarios" from the introduction. Before that, however, we would like to show a concrete example program.

We will now revisit the example of extending the λ-Calculus with numerals that we have previously seen as a *catamorphism* (in Section 3.1) and later (in Figure 1) as a *general extension pattern*, motivating the algebraic approach.

Figure 5(a) shows the λ-Calculus as a Banana Algebra *language* constant (with standard whitespace, as defined by: "`$ = [ \n\t\r]*`"). Figure 5(b) defines the transformation from the λ-Calculus extended with numerals to the core calculus (cf., Figure 1). First, the contents of the file "`lambda.l`"

```
{
   $        =  [ \n\t\r]*       ;
   Id       =  [a-z]+           ;

   exp.var  :  Id               ;
   exp.lam  :  "\\" Id "." exp  ;
   exp.app  :  "(" exp exp ")"  ;
}
```

(a) Language: λ-Calculus (with standard whitespace definition: "`[ \n\t\r]*`").

```
let l = "lambda.l"
in let ln = "lambda-num.l"
in letx ln2l =
    (| ln -> l [exp -> exp]

       exp.zero  =  '\z.z'        ;
       exp.succ  =  '\s.<1>'      ;
       exp.pred  =  '(<1> \z.z)'  ;
    |)
in ln2l + idx(l)
```

(b) Transformation: λ-Calculus extended with numerals to core λ-Calculus (cf. Fig 1).

Fig. 5. Banana Algebra example programs: a *language* and a *transformation*.

(which we assume to contain the constant in Figure 5(a)) is loaded and bound to the Banana Algebra variable, l in the rest of the program. Then, in that program, ln is bound to the language containing the extension (assumed to reside in the file "`lambda-num.l`"). After this, $ln2l$ is bound to the constant transformation that transforms the numeral extension to the core λ-Calculus. Finally, that constant transformation is added to `idx(`l`)` which is the identity transformation on the λ-Calculus.

Similarly, The Banana Algebra Tool can be used to extend Java with lots of syntactic constructions which can be desugared into Java itself; e.g., `for-each` control structures, enumeration declarations, design patterns templates, and so on. Here, we will give only one simple example of a Java extension; the `repeat-until` of Figure 3(c):

```
let java = "java.l"
in let repeat  =  { Stm.repeat  :  "repeat" Stm "until" "(" Exp ")" ";" ; }
in letx repeat2java =
   (| repeat -> java [Stm -> Stm, Exp -> Exp]
      Stm.repeat  =  'do <1> while (!(<2>));'  ;
   |)
in repeat2java + idx(java)
```

Although the Java grammar is big ("`java.l`" is a standard 575-line context-free grammar for Java), the `repeat-until` transformation is only seven lines.

More ambitiously, The Banana Algebra Tool may used to embed entire DSLs into a host language. We have used the tool to embed standard SQL constructions into the <bigwig> [4] language; e.g., the "`select-from-where`" construction may be captured by the following simple transformation:

```
stm.select = 'factor(<2>) { if (<3>) { return # \+ (<1>); } }' ;
```

Once defined, languages and transformations can all be added, composed, or otherwise put together. Thus, a programmer can use the tool to essentially tailor his own macro-extended language; e.g., "`(java \ while) + sql`".

Relying on the existence of the tool, we have used the tool on itself to add more operators to the algebra. We can easily extend the Banana Algebra with an *overwrite* operator "`<<`" on languages and transformations (defined

in terms of the core algebra):

$$\begin{array}{rcl} [\![L_1 \texttt{ << } L_2]\!]_L & = & (L_1 \texttt{ \textbackslash } L_2) \texttt{ + } L_2 \\ [\![X_1 \texttt{ << } X_2]\!]_X & = & (X_1 \texttt{ \textbackslash src(}X_2\texttt{)) + } X_2 \end{array}$$

To put the algebraic and incremental development approach to the test, we have built an entire existing functional language "Fun" (used in an undergraduate course on teaching functional programming at Aarhus University and Aalborg University). The language extends the λ-Calculus with *arithmetic*, *lists*, *pairs*, *local definitions*, *numerals* in terms of arithmetic, *signed arithmetic* in terms of booleans and pairs, *fixed-point iterators* in terms of local definitions, *types* in terms of arithmetic and pairs. The entire language is specified incrementally using 245 algebraic operators (i.e., 58 constant languages, 51 language inclusions, 28 language additions, 23 language variables, 17 constant transformations, 17 transformation additions, 14 transformation inclusions, 10 local definitions, 9 identity transformations, 8 compositions, 4 language restrictions, 4 transformation variables, and 2 source extractions). The entire transformation reduces to a constant (constructive catamorphism) transformation of size 4MB. (For more on this transformation, we refer to [1].)

7 Related Work

Our work shares many commonalities and goals with that of *syntax macros*, *source transformation systems*, and *catamorphisms* (from a category theory perspective) the relation to which will be outlined below.

Syntax macros [6,21] provide a means to *unidirectionally extend* a "host language" on top of which the macro system is hard-wired. Extension by syntactic macros corresponds to having control over only "step iii)" of Figure 1 (some systems also permit limited control over what corresponds to "step ii)"). By contrast, our algebraic approach can be used to extend the syntax of *any* language or transformation; and not just in one direction—extensions may be achieved through addition, composition, or otherwise modular assembly of other previously defined languages or transformations. Uni-directional extension is just one form of incremental definition in our algebraic approach.

The work on *extensible syntax* [9] improves on the definition flexibility in providing a way of defining grammars *incrementally*. However, it supports only three general language operations: extension, restriction, and update.

Compiler generator tools, such as Eli [12], Elan [2], Stratego/XT [8], ASF+SDF [18], TXL [10], JastAdd [13], and Silver [22] may all be used for source-to-target language transformation. They all have wider ambitions than our work, supporting specifications of full-scale compilers, many including static and dynamic semantics as well as Turing Complete computation on ASTs of the source language which obviously precludes our level of safety guarantees.

Although many of the tools support modular language development, none of them provide an algebra on top of their languages and transformations.

Systems based on *attribute grammars* (e.g., Eli, JastAdd, and Silver) may be used to indirectly express source-to-target transformations. This can be achieved through Turing Complete computation on the AST of the source language which compute terms of the target language in a downward or upward fashion (through *synthesized* and *inherited* attributes), or combinations thereof. In contrast, catamorphisms are restricted to upward inductive recombination of target ASTs. Our transformations could easily be generalized to also construct target AST downwards, by simply allowing catamorphisms to take target typed *AST arguments* (as detailed in [7], p. 17). This corresponds to a notion of *anamorphisms* and *hylomorphisms*, but would compromise compile-time elimination of composition (since anamorphisms and catamorphisms in general cannot be fusioned into one transformation, without an intermediate step).

Systems based on *term rewriting* (e.g., Elan, TXL, ASF+SDF, and Stratego/XT) may also be used to indirectly express source-to-target transformations. However, a transformation from language S to T has to be *encoded* as a rewriting working on terms of combined type: $S \cup T$ or $S \times T$. Although the tools may syntactically check that each rewriting step respects the grammars, the formalism comes with three kinds of termination problems which cannot be statically verified in either of the tools; a transformation may: i) never terminate; ii) terminate too soon (with unprocessed source terms); and, iii) be capable of producing a forest of output ASTs which means that is the responsibility of the programmer to ensure that the end result is one single output term. To help the programmer achieve this, rewriting systems usually offer control over the rewriting strategies.

In order to issue strong safety guarantees, in particular termination, we clearly sacrifice expressibility in that the catamorphisms are *not* able to perform Turing Complete transformations. However, previous work using constructive catamorphisms for syntactic transformations (e.g., Metafront [7] and XSugar [5]) indicate that they are sufficiently expressive and useful for a wide range of applications.

Of course, catamorphisms may be mimicked by disciplined style of functional programming, possibly aided by traversal functions automatically synthesized from datatypes [15], or by libraries of combinators [17]. However, since within a general purpose context, it cannot provide our level of safety guarantees and would not be able to compile-time factorize composition (although the functional techniques *deforestation/fusion* [20,11,19] may—in some instances—be used to achieve similar effects).

There exists a body of work on catamorphisms in a category theoretical setting [14,16]. However, these are theoretical frameworks that have not been turned into practical tool implementations supporting the notion of addition on languages and transformations which plays a crucial role in the extension

pattern of Figure 1 and many of the examples.

8 Conclusion

The algebraic approach offers via 16 operators a *simple*, *incremental*, and *modular* means for specifying syntactic language extensions through algebraic composition of previously defined languages and transformations. The algebra comes "for free" in that any algebraic transformation term can be statically *reduced* to a constant transformation without compromising the strong *safety* and *efficiency* properties offered by catamorphisms.

The tool may be used by: 1) programmers to extend existing languages with their own macros; 2) developers to embed DSLs in host languages; 3) compiler writers to implement only a small core language (and specify the rest externally as extensions); and 4) developers and teachers to build multi-layered languages. The Banana Algebra Tool is available—as 3,600 lines of O'Caml code—along with examples from its homepage:

`[ http://www.itu.dk/people/brabrand/banana-algebra/ ]`

Acknowledgments

The authors would like to acknowledge Kevin Millikin, Mads Sig Ager, Per Graa, Kristian Stvring, Anders Mller, Michael Schwartzbach, and Martin Sulzmann for useful comments and suggestions.

References

[1] Jacob Andersen and Claus Brabrand. Syntactic language extension via an algebra of languages and transformations. ITU Technical Report. Available from: http://www.itu.dk/people/brabrand/banana-algebra/, 2008.

[2] P. Borovansky, C. Kirchner, H. Kirchner, P. Moreau, and C. Ringeissen. An overview of elan. In *Second Intl. Workshop on Rewriting Logic and its Applications*, volume 15, 1998.

[3] Claus Brabrand, Robert Giegerich, and Anders Møller. Analyzing ambiguity of context-free grammars. In *Proc. 12th International Conference on Implementation and Application of Automata, CIAA '07*, July 2007.

[4] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The `<bigwig>` project. *ACM Transactions on Internet Technology*, 2(2):79–114, 2002.

[5] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Dual syntax for XML languages. *Information Systems*, 33(4), June 2008. Earlier version in Proc. 10th International Workshop on Database Programming Languages, DBPL '05, Springer-Verlag LNCS vol. 3774.

[6] Claus Brabrand and Michael I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and semantics-based Program Manipulation, PEPM'02*. ACM, 2002.

[7] Claus Brabrand and Michael I. Schwartzbach. The metafront system: Safe and extensible parsing and transformation. *Science of Computer Programming Journal (SCP)*, 68(1):2–20, 2007.

[8] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

[9] Luca Cardelli, Florian Matthes, and Martin Abadi. Extensible syntax with lexical scoping. SRC Research Report 121, 1994.

[10] J.R. Cordy. Txl - a language for programming language tools and applications. In *Proceedings of ACM 4th International Workshop on Language Descriptions, Tools and Applications (LDTA'04)*, pages 1–27, April 2004.

[11] Joo Paulo Fernandes, Alberto Pardo, and Joo Saraiva. A shortcut fusion rule for circular program calculation. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 95–106. ACM, 2007.

[12] Robert W. Gray, Steven P. Levi, Vincent P. Heuring, Anthony M. Sloane, and William M. Waite. Eli: a complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–130, 1992.

[13] Grel Hedin and Eva Magnusson. Jastadd - a java-based system for implementing frontends. In *Electronic Notes in Theoretical Computer Science*, volume 44(2). Elsevier Science Publishers, 2001.

[14] Richard B. Kieburtz and Jeffrey Lewis. Programming with algebras. In *Advanced Functional Programming, number 925 in Lecture Notes in Computer Science*, pages 267–307. Springer-Verlag, 1995.

[15] R. Lämmel, J. Visser, and J. Kort. Dealing with Large Bananas. In J. Jeuring, editor, *Proceedings of WGP'2000, Technical Report, Universiteit Utrecht*, pages 46–59, July 2000.

[16] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings 5th ACM Conf. on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA, 26–30 Aug 1991*, volume 523, pages 124–144. Springer-Verlag, Berlin, 1991.

[17] S. Doaitse Swierstra, Pablo R. Azero Alcocer, and Joo Saraiva. Designing and implementing combinator languages. In *Third Summer School on Advanced Functional Programming, volume 1608 of LNCS*, pages 150–206. Springer-Verlag, 1999.

[18] M. G. J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju,

E. Visser, and J. Visser. The ASF+SDF meta-environment: a component-based language development environment. In *Proc. Compiler Construction 2001*. Springer-Verlag, 2001.

[19] Janis Voigtländer. Semantics and pragmatics of new shortcut fusion rules. In Jacques Garrigue and Manuel Hermenegildo, editors, *Proc. Functional and Logic Programming*, volume 4989 of *LNCS*, pages 163–179. Springer-Verlag, April 2008.

[20] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:344–358, 1990.

[21] Daniel Weise and Roger F. Crew. Programmable syntax macros. In *Programming Language Design and Implementation (PLDI)*, pages 156–165, 1993.

[22] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an extensible attribute grammar system. *Electronic Notes in Theoretical Computer Science*, 203(2):103–116, 2008.

A Semantics of the algebra

We will now exploit the aforementioned self-containedness property and give a *big-step reduction semantics* for the algebra capable of reducing any language expression, L, to a constant language (context-free grammar), l; and any transformation expression, X, to a constant transformation (constructive catamorphism), $x = \{\!|\, l_s \rightarrow l_t \; [\tau] \; c \,|\!\}$.

Let EXP_L denote the set of all language expressions from the syntactic category, L; and let EXP_X denote the set of all transformation expressions from the syntactic category, X. Also, we take VAR to be the set of all variables. We define environments in a straightforward way:

$$ENV_L : VAR \rightarrow EXP_L \qquad ENV_X : VAR \rightarrow EXP_X$$

The reduction semantics for the algebra of languages is defined by the relation $\Downarrow_L \subseteq ENV_L \times ENV_X \times EXP_L \times EXP_L$ (cf. Figure 1(a)). We will use the syntax "$\alpha, \beta \vdash L \Downarrow_L l$" as a shorthand for "$(\alpha, \beta, L, l) \in \Downarrow_L$". Similarly, the reduction semantics for the algebra of transformations is defined by the relation $\Downarrow_X \subseteq ENV_L \times ENV_X \times EXP_X \times EXP_X$ (cf. Figure 1(b)). Again, we will use the short-hand syntax "$\alpha, \beta \vdash X \Downarrow_X x$" instead of "$(\alpha, \beta, X, x) \in \Downarrow_X$".

Note that the reduction semantics in Figure A.1 uses a range of operators ($\vdash_{wfl}$, $\sim_l$, $\oplus_l$, $\ominus_l$, $\sqsubseteq_l$, $\vdash_{wfx}$, $\sim_x$, $\oplus_x$, $\ominus_x$, id_τ, id_c) which all operate on the level below that of the algebra; i.e., on constant languages (context-free grammars) and transformations (constructive catamorphisms). They can all be defined either at a *concrete* or *abstract* syntactic level. We refer to [1], for a formal specification of these lower-level operators in terms of abstract syntax.

$$[\text{CON}]_L \ \frac{}{\alpha, \beta \vdash l \Downarrow_L l} \quad \vdash_{wfl} l$$

$$[\text{VAR}]_L \ \frac{}{\alpha, \beta \vdash v \Downarrow_L \alpha(v)}$$

$$[\text{RES}]_L \ \frac{\alpha, \beta \vdash L \Downarrow_L l \qquad \alpha, \beta \vdash L' \Downarrow_L l'}{\alpha, \beta \vdash L \ \backslash \ L' \Downarrow_L l \ominus_l l'}$$

$$[\text{ADD}]_L \ \frac{\alpha, \beta \vdash L \Downarrow_L l \qquad \alpha, \beta \vdash L' \Downarrow_L l'}{\alpha, \beta \vdash L + L' \Downarrow_L l \oplus_l l'} \quad l \sim_l l'$$

$$[\text{SRC}]_L \ \frac{\alpha, \beta \vdash X \Downarrow_X \langle\!| l_s \rightarrow l_t \ [\tau] \ c |\!\rangle}{\alpha, \beta \vdash \texttt{src (} X \texttt{)} \ \Downarrow_L l_s}$$

$$[\text{TGT}]_L \ \frac{\alpha, \beta \vdash X \Downarrow_X \langle\!| l_s \rightarrow l_t \ [\tau] \ c |\!\rangle}{\alpha, \beta \vdash \texttt{tgt (} X \texttt{)} \ \Downarrow_L l_t}$$

$$[\text{LET}]_L \ \frac{\alpha, \beta \vdash L \Downarrow_L l \qquad \alpha[v \mapsto l], \beta \vdash L' \Downarrow_L l'}{\alpha, \beta \vdash \texttt{let } v\texttt{=}L \texttt{ in } L' \Downarrow_L l'}$$

$$[\text{LETX}]_L \ \frac{\alpha, \beta \vdash X \Downarrow_X x \qquad \alpha, \beta[w \mapsto x] \vdash L' \Downarrow_L l'}{\alpha, \beta \vdash \texttt{letx } w\texttt{=}X \texttt{ in } L' \Downarrow_L l'}$$

(a) Semantics for the algebra of languages.

$$[\text{CON}]_X \ \frac{\alpha, \beta \vdash L_s \Downarrow_L l_s \qquad \alpha, \beta \vdash L_t \Downarrow_L l_t}{\alpha, \beta \vdash \langle\!| L_s \rightarrow L_t \ [\tau] \ c |\!\rangle \Downarrow_X \langle\!| l_s \rightarrow l_t \ [\tau] \ c |\!\rangle} \quad \vdash_{wfx} \langle\!| l_s \rightarrow l_t \ [\tau] \ c |\!\rangle$$

$$[\text{VAR}]_X \ \frac{}{\alpha, \beta \vdash w \Downarrow_X \beta(w)}$$

$$[\text{RES}]_X \ \frac{\alpha, \beta \vdash X \Downarrow_X x \qquad \alpha, \beta \vdash L \Downarrow_L l}{\alpha, \beta \vdash X \ \backslash \ L \Downarrow_X x \ominus_x l}$$

$$[\text{ADD}]_X \ \frac{\alpha, \beta \vdash X \Downarrow_X x \qquad \alpha, \beta \vdash X' \Downarrow_X x'}{\alpha, \beta \vdash X + X' \Downarrow_X x \oplus_x x'} \quad x \sim_x x'$$

$$[\text{COMP}]_X \ \frac{\alpha, \beta \vdash X \Downarrow_X \langle\!| l_s \rightarrow l_t \ [\tau] \ c |\!\rangle \qquad \alpha, \beta \vdash X' \Downarrow_X \langle\!| l'_s \rightarrow l'_t \ [\tau'] \ c' |\!\rangle}{\alpha, \beta \vdash X' \circ X \Downarrow_X \langle\!| l_s \rightarrow l'_t \ [\tau' \circ \tau] \ c' \circ_c c |\!\rangle} \quad l_t \sqsubseteq_l l'_s$$

$$[\text{IDX}]_X \ \frac{\alpha, \beta \vdash L \Downarrow_L l}{\alpha, \beta \vdash \texttt{idx (} L \texttt{)} \ \Downarrow_X \langle\!| l \rightarrow l \ [id_\tau(l)] \ id_c(l) |\!\rangle}$$

$$[\text{LET}]_X \ \frac{\alpha, \beta \vdash L \Downarrow_L l \qquad \alpha[v \mapsto l], \beta \vdash X' \Downarrow_X x'}{\alpha, \beta \vdash \texttt{let } v\texttt{=}L \texttt{ in } X' \Downarrow_X x'}$$

$$[\text{LETX}]_X \ \frac{\alpha, \beta \vdash X \Downarrow_X x \qquad \alpha, \beta[w \mapsto x] \vdash X' \Downarrow_X x'}{\alpha, \beta \vdash \texttt{letx } w\texttt{=}X \texttt{ in } X' \Downarrow_X x'}$$

(b) Semantics for the algebra of transformations.

Fig. A.1. Semantics of the algebra

Formalizing Homogeneous Language Embeddings

Tony Clark tony.clark@tvu.ac.uk

Thames Valley University, St. Mary's Road, Ealing, London, W5 5RF, United Kingdom

Laurence Tratt laurie@tratt.net

Bournemouth University, Poole, Dorset, BH12 5BB, United Kingdom

Abstract

The cost of implementing syntactically distinct Domain Specific Languages (DSLs) can be reduced by homogeneously embedding them in a host language in cooperation with its compiler. Current homogeneous embedding approaches either restrict the embedding of multiple DSLs in order to provide safety guarantees, or allow multiple DSLs to be embedded but force the user to deal with the interoperability burden. In this paper we present the μ-calculus which allows parameterisable language embeddings to be specified and analysed. By reducing the problem to its core essentials we are able to show how multiple, expressive language embeddings can be defined in a homogeneous embedding context. We further show how variant calculi with safety guarantees can be defined.

1 Introduction

Domain Specific Languages (DSLs) are mini languages used to aid the implementation of recurring problems. What identifies a particular language as being a 'DSL' is partly subjective; intuitively, it is a language with its own syntax and semantics, but which is smaller and less generic than a typical GPL such as Java. The DSL premise is simple: a one off, up front, cost allows classes of systems to be created at low cost and in a reliable and maintainable fashion [16].

DSLs have a long history, although they have often gone by different names [1]. Traditional, widely used DSLs such as the UNIX `make` program and the `yacc` parsing system have been implemented as stand-alone systems, which are effectively cut-down programming language compilers and virtual machines rolled into one; the associated implementation costs and lack of practical reusability have hampered the creation of DSLs [10]. An alternative approach to stand-alone implementation is *embedding*, where a DSL is 'hosted' within a host programming language; in other words, the host languages' syntax is extended with the DSLs syntax. A simple example of such embedding is an SQL DSL; by using a DSL instead of an

external database library one gains several advantages such as the static detection of SQL syntax errors and the safe insertion of external values into SQL statements. Language extension has been a goal of language researchers for several decades (see e.g. [12]) but most early efforts were unable to prevent unintended and unwanted interactions between languages and their extensions [5]. Later approaches, though largely theoretical, did show that certain forms of language extension could avoid such problems [3,4]. More recently, DSL embedding approaches such as Stratego [2], XMF [6,7], Converge [17], Metalua [9], and others (e.g. [14,8,13]) have shown that this is a viable approach.

DSL embedding techniques can be classified as either *heterogeneous* or *homogeneous* [17]. Put simply, heterogeneous embedding (e.g. Stratego) is when the system used to compile the host language, and the system used to implement the embedding are different (note that this does not imply that the host language must be different than the language used to implement the embedding). In contrast, homogeneous embedding (e.g. Converge, Metalua, XMF) uses the language's compiler to compile the host language and to facilitate DSL embedding. Heterogeneous embedding has the advantage that it can be applied to any host language and any embedded language. However this means that heterogeneous embedding systems generally have little or no idea of the semantics of the languages they are embedding into, meaning that such techniques are hard to scale up [17]. Furthermore heterogeneous techniques typically assume that a single DSL is embedded into a single host language: multiple distinct DSLs must be manually welded together in order to create a single heterogeneous embedding which does not suffer from syntax errors. Homogeneous embedding, however, is inherently limited to a specific host language, but is typically able to offer greater guarantees about the safety of the resulting embedding, allowing larger and more complex DSLs to be embedded. Homogeneous embedding also places no conceptual restrictions on embedding multiple DSLs in one host language, or having DSLs interleaved within each other.

In practice, current homogeneous embedding technologies limit the extent to which multiple DSLs can be embedded without resorting to unwieldy hacks [9]. For example, Metalua allows multiple DSLs to be embedded within it, but requires manipulation of the global parser; no guarantees are made that different extensions will co-exist peacefully, or even that individual extensions are well-formed. Converge, on the other hand, allows multiple DSLs to co-exist and enforces reasonable safety guarantees but does so by making DSLs unpleasantly syntactically distinct, and making embedding DSLs within each other extremely difficult.

We believe that the distance between the conceptual promise and current practical realities of homogeneous embedding are in large part because of a lack of understanding of the underlying theory of language embedding in a homogeneous setting. In this paper we present the μ-calculus for specifying and analysing homogeneous language embedding. The calculus extends the λ-calculus with facilities for defining and using language embeddings, allowing parameterisable language definitions to be scoped to portions of a source file, and to be nested arbitrarily

within each other.

This paper is structured as follows: Section 2 defines the syntax and semantics of the μ-calculus and shows how to precisely define an embedded language; Section 3 describes different categories of language embedding and how they are encoded in the calculus; Section 4 shows how a language with more than one embedding can be defined using the calculus and how safety criteria are represented; finally, Section 5 concludes the paper with an analysis and discussion of further work.

2 μ: A Language Embedding Calculus

The μ-calculus is an extension of the λ-calculus that supports embedded languages. New languages can be added to μ by defining their syntax, dynamic semantics, and their relationship to the execution context of the host language.

The definition and subsequent use of an embedded language takes the form of a standard structure within the μ-calculus. This section defines the calculus and defines how languages are embedded within it. It is structured as follows: Section 2.1 defines the syntax of the calculus and provides an example of its use; languages are embedded in terms of their abstract syntax, Section 2.2 defines a data type that represents μ-calculus abstract syntax; Section 2.3 defines the semantics of the μ-calculus by embedding it within itself.

2.1 Overview

The syntax of the μ-calculus is:

E ::=		expressions
	V	variables
	$\mid$ fun$(V)E$	functions
	$\mid$ EE	applications
	$\mid$ if E then E else E	conditionals
	$\mid$ (E,E,E)	language definition
	$\mid$ lang $E : T[C]$	language embedding
T ::=	$\ldots$	syntax types
C ::=	$\ldots$	raw text

μ contains the conventional λ-calculus, plus language definition and language embedding components. A language definition defines a language's semantics as an interpreter and how to embed it in the context of a host.

A language embedding allows the use of a language within the host calculus. In essence, language definitions define interpreters and a translation from a host interpreter to the embedded language's interpreter. A language embedding is a use of the definition in a context provided by the host language.

More specifically, the language definition triple $(eval, load, unload)$ defines: an evaluator, *eval*, which evaluates the language in terms of its state; a loader, *load*,

that maps from the host language state to the embedded language state; and an unloader, *unload*, that translates the embedded state back into a host state. The following is an overview of how the calculus might be used:

```
// Define an embedded SQL-like language...
type SQL = ... // type definition for SQL.
let sql = (evalSQL,loadSQL,unloadSQL)
            where
              evalSQL = ...
              loadSQL = ...
              unloadSQL = ...
// Define an embedded HTML-like language...
type HTML = ... // type definition for HTML.
let html = (evalHTML,loadHTML,unloadHTML)
             where ...
// Use the two embedded language definitions.
// Perform database queries to produce all the
// (name,age) pairs for adults...
let results =
     lang sql:SQL[SELECT name,age from Customer WHERE age > 18]
in // Produce the HTML table showing the results...
   lang html:HTML[
     <TABLE>
       for name,age in results do
         <TR>
           <TD> name </TD>
           <TD> age  </TD>
         </TR>
     </TABLE>
   ]
```

2.2 *Abstract Syntax Types*

The type definition for the μ-calculus is as follows:

```
type Exp(T) =
       Var(String)
     | Lambda(String,Exp(T))
     | Apply(Exp(T),Exp(T))
     | If(Exp(T),Exp(T),Exp(T))
     | Lang(T)
```

The type definition is parameterized with respect to the type of embedded languages: `T`. If a single language L is embedded then the resulting type is `Exp(L)`. If more than one language is embedded, then we use a disjoint type combinator to express the type of the resulting language: `Exp(L + M)`. Finally, a fix-point operator `Y` can be used to construct a type. For example, `Y(Exp)` is the type of languages constructed by embedding the μ-calculus in itself.

2.3 *Semantics*

The semantics of the μ-calculus is defined as a language embedding as follows. The evaluator for the calculus can be any suitable definition. To maximise the potential for future extension we implement the evaluator as a state machine. This

ensures that any embedded language has access to all data *and* control structures of the language. The canonical state machine for a λ-calculus is the SECD machine [11]. The type definition is as follows:

```
type State = ([Value],Env,[Instr],State) | Empty
type Env = String->Value
type Instr = Exp(T) | App | If(Exp(T),Exp(T))
type Value = Basic | Closure | State
type Closure = (String,Env,Exp(T))
```

The evaluator is a state transition function. It is supplied with a current machine state, performs a single transition, producing a new state. It is also supplied with another state transition function `eval` to which it supplies the new state. By supplying `eval`, the basic μ-calculus evaluator can be extended:

```
evalExp(eval)(s) =
  case s of
    ([v],_,[],Empty)          -> s
    (s,e,Var(n):s,d)          -> eval(e(n):s,e,c,d)
    (s,e,Lambda(n,b):c,d)     -> eval((n,e,b):s,e,c,d)
    (s,e,Apply(o,a):c,s)      -> eval(s,e,a:o:App:c,d)
    (s,e,If(f,g,h):c,s)       -> eval(s,f:If(g,h):c,d)
    (true:s,e,If(g,h):c,d)    -> eval(s,e,g:c,d)
    (false:s,e,If(g,h):c,d)   -> eval(s,e,h:c,d)
    ((n,e',b):v:s,e,App:c,d) -> eval([],e'[n->v],[b],(s,e,c,d))
    (R:():s,e,App:c,d)        -> eval((s,e,c,d):s,e,c,d)
    (I:v:s,e,App:c,d)         -> eval(v)
    ([v],_,[],(s,e,c,d))      -> eval(v:s,e,c,d)
  end
```

The case statement above defines a state transition machine where states have the form `(S,E,C,D)` where `S` is a stack of intermediate results implemented as a list, `E` is an environment of variable bindings, `C` is a sequence of machine instructions, and `D` is a machine resumption state. The syntax of the above language is largely self explanatory. Lists are represented as either standard square bracketed sequences of elements, or cons pairs `h:t`. A name `n` is looked up in an environment `e` with `e(n)`, and a value `v` is added to the environment with `e[n->v]`. The builtin operators `R` and `I` are used to *reify* and *intern* machine states. Assuming the existence of a parsing mechanism that indexes on the type of a language definition, the expression `lang(e,l,u):t[c]` is equivalent to the following expression:

```
I(newState)
  where newState = u(termState,initialState)
    where termState = e(startState)
      where startState = l(initialState,parse(t)(c))
        where initialState = R()
```

The expression above uses a parser that is indexed on the type `t` of the embedded language. It is outside the scope of this paper to analyse how parsing mechanisms can be supported by the μ-calculus; however, the parser produces values of the appropriate abstract syntax type.

The initial state is created by reifying the current μ-context. The initial state is supplied to the loader `l` together with the abstract syntax to produce a starting state

for the embedded language evaluator. The starting state is supplied to the evaluator `e` to produce a terminal state. The terminal state along with the original initial state is supplied to the unloader to produce a new host language state. The new state is then interned by supplying it to the host language interpreter.

If we embed μ in itself then the load and unload operations are identity. Therefore the definition of μ is:

```
let Mu = Y(Exp)
let evalMu = Y(evalExp)
let loadMu((s,e,c,d),x) = (s,e,x:s,d)
let unloadMu(s,_) = s
let muL = (evalMu,loadMu,unloadMu)
```

Now we can write programs that arbitrarily nest the calculus in itself (given suitable sugarings for infix operators):

```
fun(x) lang muL:Mu[fun(y) lang muL:Mu[x + y]]
```

In conclusion a language definition consists of: a parser for the language (which is not considered further by this paper); a data type for the language abstract syntax; a context data type for the language evaluator; an evaluator that processes the context; a loader that maps from host contexts to embedded contexts; an unloader that maps from embedded contexts to host contexts.

3 Categories and Styles of Language Embedding

The μ-calculus can be used to embed any language l within a host h. The intended usage is that h is defined as a language within μ and then l is defined within h. The approach supported by μ forces a precise definition of *how* l is embedded within h including any safety criteria. μ allows the embedding to be analysed prior to implementation.

There are a number of different types of language embedding. Some embeddings are *functional* because uses of the language denote values; some are non-functional because they modify the host language context; many language embeddings require that the bindings from the host language are transferred to the embedded language; some embedded languages require private state and some require that the state can be communicated to other embedded languages.

The μ-calculus can be used to define what we term *uniform* and *ad-hoc* languages. Uniform languages are those that extend the μ-calculus interpreter, and thus allow languages to be embedded inside them using the standard μ-calculus techniques. Ad-hoc languages are those that define an arbitrary interpreter; while it is still possible to embed other languages within them, this must be done manually on a case-by-case basis. This section provides examples of different categories of language embedding using the μ-calculus.

3.1 A Simple Extension

One of the simple programming language-like features not found in μ is a *let-binding*. In this section we show how this can be added as a language embedding to μ. The type for expressions in the language `Let` is defined by extending the basic expression language:

```
type LetExp(T) = Let(String,Let(T),Let(T)) | Exp(T)
type Let = Y(LetExp)
```

An evaluator for `Let` is defined by extending the evaluator for the basic calculus:

```
let evalLetExp(eval)(s) =
  case s of
    (s,e,Let(n,x,b):c,d) -> eval(s,e,x:Let(n,b):c,d)
    (v:s,e,Let(n,b):c,d) -> eval([],e[n->v],[b],(s,e,c,d))
    else evalExp(eval)(s)
  end
```

The `Let` language definition follows the same structure as the `Mu` language definition: the load and unload operations are essentially identity mappings:

```
type Let = Y(LetExp)
let evalLet = Y(evalLetExp)
let loadLet((s,e,c,d),x) = (s,e,x:c,d)
let unoadLet(s,_) = s
let letL = (evalLet,loadLet,unloadLet)
```

Now, the let language can be used when it is embedded in the basic calculus which is now of type `Exp(Let)`:

```
fun(x) lang letL:Let[let y = x + 1 in y ]
```

Note also that because the `Let` language is uniform, it can be used as the basis for further language embeddings. This is shown in the following section.

3.2 Localized Data

A language extension is often useful when creating data structures. If the application domain is specialized then the language extension can provide abstractions that make the construction of the data *declarative* in the sense that low-level language features that are necessary to create the structure are hidden away.

The definition of a new feature for data is an example of a *functional* language embedding. Such a language is not necessarily uniform, however it does not modify the state of the host language and is used exclusively for its value.

This section provides an example of a functional embedding for implementing arrays. An array can be encoded in the μ-calculus using functions:

```
let mkArray(j) = null
let set(i,v,a) = fun(j) if i = j then v else a(j)
```

There may be many initial values when an array is created. Without a declarative language feature to achieve this, the creation will involve many nested calls to `set`. The language `Array` is used to create arrays:

```
lang letL:Let[
  let mkArray = fun(limit) lang arrayL:Array [ 0 .. limit ]
  in mkArray(100)]
```

The abstract syntax of the array language is not an extension to `Exp`:

```
type ArrayExp = (ArrayVal,ArrayVal)
type ArrayVal = Var(String) | Int
```

The evaluator for the array language only requires information about the binding context from the host language. The evaluator creates a value of type `[Value]`:

```
let evalArray((lower,upper),env) =
  letrec mkList(l,u) = if l = u then [l] else lower:mkList(l+1,u)
         deref(Var(n)) = env(n)
         deref(i) = i
  in mkList(deref(lower),deref(upper))
```

The host language cannot manipulate values of type `[Value]` (and in general for an embedded language there may be a wide variety of 'foreign' values). Therefore the array unloader must translate between the array representation and the calculus representation:

```
let transArray : [Value] -> Closure
let transArray([],i) = (j,[],[| null |])
let transArray(v:a,i) =
  (j,a->transArray(a,i+1),[| if j = <i> then <v> else a(j) |])
```

The operation `transArray` defined above uses quasi-quotes to construct and manipulate abstract syntax in terms of concrete syntax. We will assume that this language feature is available in the μ-calculus since it is simply sugar for the equivalent expression in terms of AST constructors. Quasi-quotes have been implemented in a number of languages to support syntax manipulation including Template Haskell [15], Converge and XMF.

Given the translation from arrays (lists of values) to a closure-based representation, it is possible to define the array loader and unloader:

```
let loadArray((s,e,c,d),(l,u)) = ((l,u),e)
let unloadArray(a,(s,e,c,d)) = (transArray(a,0):s,e,c,d)
```

We can now define a language that embeds both the μ-calculus and arrays into `Let`

```
type Lang(T) = Exp(Let(Exp(T) + ArrayExp)
type Array = Y(Lang)
let arrayL = (evalArray,loadArray,unloadArray)
```

The language used in the example above is defined by `Exp(Array)`.

3.3 State Modification

Not all languages are functional. A non-functional language can affect the state of the host language in some way. The impact of the language can be on the data, on the control flow or both. This section provides a simple ad-hoc language embedding that affects the state of data in the host language.

Consider the case where we want to print out a message each time a μ-calculus

function is called. The calculus does not provide any features that allow us to toggle function tracing on and off. A new language feature is required that allows the following:

```
lang traceL:Trace[ traceOn ]
... // Tracing is now on...
lang traceL:Trace[ traceOff ]
... // No more tracing
```

We will assume that there are builtin functions called `enter` and `exit` in the μ-calculus that allow functions to be traced. So a function `fun(x) b` can be traced by changing the function body to:

```
fun(x) exit(b,enter(x))
```

The language for tracing is very simple:

```
type Trace = traceOn | traceOff
```

When tracing is switched on, the embedded language makes a global change to the host state. Each closure and function expression must be modified to insert calls to the tracing functions:

```
let trace(s,e,c,d) = (trace(s),trace(e),trace(c),trace(d))
let trace(Empty) = Empty
let trace([]) = []
let trace(x:s) = trace(x):trace(s)
let trace(n->v) = n->trace(v)
let trace(n,e,b) = (n,trace(e),[| exit(<trace(b)>,enter(<n>)) |])
let trace(Var(n)) = Var(n)
let trace(Lambda(n,b) = Lambda(n,[| exit(<trace(b)>,enter(<n>)) |])
let trace(Apply(m,n)) = Apply(trace(m),trace(n))
let trace(App) = App
```

The `untrace` operator performs the changes in reverse. Now the tracing language can be defined in terms of the global modifier to the host language state:

```
let evalTrace(s,traceOn) = trace(s)
let evalTrace(s,traceOff) = untrace(s)
let loadTrace(s,t) = (s,t)
let unloadTrace(s,s') = s
let traceL = (evalTrace,loadTrace,unloadTrace)
```

The calculus with tracing in is defined by `Y(Lang)` where:

```
type Lang(T) = Exp(Exp(T) + Trace)
```

3.4 Control Flow

The previous section defines a non-functional embedding that influences the structure of data in the host language. Another form of non-functional embedding affects the control flow of the host language. This is only possible if the embedded language has access to the complete state of the host language. It can be achieved by passing continuations to the embedded language, however this makes it difficult to define transformations on the state. Instead, if evaluators are defined in terms of transition machines then embedded languages have access to the required informa-

tion in an appropriate format.

Suppose that we want a new language construct that aborts the program under a given condition:

```
lang letL:Let[
  let x = f(100)
  in lang abortL:Abort[ stop if(x > 100) ]]
```

The condition under which the program aborts is written in the μ-calculus, therefore the language is defined as:

```
type Abort(T) = Exp(T)
type Lang(T) = Exp(Let(Exp(T) + Abort(Exp(T))))
```

Note that in the example above the type `Abort` is a synonym for `Exp`, however the parser will use a different constructor to tag the result of parsing the embedded language.

The evaluator for the embedded language must extend that of the μ-calculus:

```
let evalAbort(eval)(expState,trueState,falseState) =
  trueState    when eval(expState) = ([true],_,_,_)
  falseState   otherwise
```

The loader and unloader for the `Abort` language are defined as follows:

```
let loadAbort((s,e,c,d),x) =
  (([],e,[x],Empty),(s,e,c,d),([error],[],[],Empty)
let unloadAbort(s,_) = s
```

The language for `Abort` is defined:

```
let abortL = (evalAbort(Y(evalExp)),loadAbort,unloadAbort)
```

3.5 Private State

Previous examples have shown how the identifiers that are in scope within the host language can be passed down to an embedded language. In general, this is achieved by the loader for the language passing the current environment to the evaluator for the embedded language.

Multiple occurrences of the same embedded language may require access to shared data. This can be achieved through binding in the host language and making the variables in scope available to the embedded language. However, this is not always desirable since the embedded language must know the names of the variables that hold the values of its state. In general, it is unsafe to rely on the use of particular variable names to pass information from one language to another.

Another option is to encode the state required by the embedded language as part of the evaluation state of the host language. This requires the host language state to be extended. This section shows how the state is extended.

Consider a language `Secret` that has a single boolean flag. Each occurrence of the embedded language may choose to toggle the flag or print it out:

```
type Secret = Toggle | Print
```

There are two new elements of state required by the secret language: the flag; the

stream of outputs. Therefore, the state of the host calculus becomes: `(s,e,c,d,f,o)` where `f` is a boolean flag and `o` is a sequence of boolean representing the output of the program.

```
type SecretState = ([Value],Env,[Instr],SecretState,Bool,[Bool])
```

We have already defined `evalExp` and do not want to change it to reflect the extended state. The solution is to wrap the definition of `evalExp` with a new definition that *lifts* the signature from `State -> State` to `SecretState -> SecretState`. This is defined as follows:

```
let evalExp'(eval)(s,e,c,d,f,o) = evalExp(eval')(s,e,c,d)
      where eval'(s,e,c,d) = eval(s,e,c,d,f,o)
```

The evaluator for `Secret` is simple:

```
let evalSecret((f,o),Toggle) = (!f,o)
let evalSecret((f,o,Print) = (f,f:o)
```

The language definition for `Secret` is:

```
let loadSecret((s,e,c,d,f,o),x) = ((f,o),x)
let unloadSecret((f,o),(s,e,c,d,_,_)) = (s,e,c,d,f,o)
let secretL = (evalSecret,loadSecret,unloadSecret)
```

The calculus with `Secret` embedded within `Let` can be defined as `Y(Lang)`:

```
type Lang(T) = Exp(Let(Exp(T) + Secret))
```

3.6 New Binding Schemes

The μ-calculus is statically scoped. Dynamic scoping allows variables to be bound to values such that they are available anywhere in the program during the evaluation of a given expression. Dynamic scoping is useful to capture the situation where a variable would need to be passed to many operations as an argument. Common Lisp is an example of a language that provides both static and dynamic scoping. The following example shows how a dynamic binding scheme works:

```
lang letL:Let[
  let add = fun(x) x + y
  in lang dynL:Dyn[dyn y = 100 in add(20)]]
```

The function `add` takes a formal parameter `x` and adds it to `y` which is currently not in scope. The embedded language `Dyn` binds `y` and then calls `add` supplying it with `20`. The result of the program is `120`.

The dynamic binding language requires a new type of environment for dynamic variables. Just as `Let` extends and contracts the static environment, `Dyn` extends and contracts the dynamic environment. However, the evaluator for the basic `Exp` language cannot reference a new type of environment since the state is fixed.

The solution is to introduce a new state element for a dynamic environment and to merge the static and dynamic environments when the `Dyn` interpreter passes control to the `Exp` interpreter. When the state is returned by the `Exp` interpreter, the dynamic environment is extracted and replaced into the `Dyn` state. The type `Dyn` is defined:

```
type Dyn(T) = DLet(String,Dyn(T),Dyn(T)) | Exp(T)
```

The evaluator for `Dyn` is:

```
let evalDyn(eval)(s) =
  case s of
    (s,e,y,DLet(n,x,b):c,d) -> eval(s,e,y,x:DLet(n,b):c,d)
    (v:s,e,y,DLet(n,b):c,d) -> eval([],e,y[n->v],[b],(s,e,y,c,d))
    ([v],_,_,[],(s,e,y,c,d)) -> eval(v:s,e,y,c,d)
    else evalExp'(eval)(s)
  end
```

When the evaluator `evalDyn` is called it checks for dynamic binding expressions at the head of the control. The dynamic binding expression evaluates the value-part and extends the dynamic environment in the machine state. The evaluator `evalExp` must be *lifted* to take account of the extra state component:

```
let evalExp'(eval)(s,e,,y,c,d) = evalExp(eval')(s,y + e,c,d)
      where eval'(s,e,c,d) = eval(s',e',y,c,d')
        where (s',e',c,d') = (s,e,c,d)/y
```

The definition of `evalExp'` given above merges the dynamic and lexical environments when it calls `evalExp`. When the environments are merged. the lexical environment always takes precedence allowing lexically bound variables to shadow the dynamic variables. Since the lexical environment always shadows the dynamic environment, it is possible to remove the dynamic environment from the resulting state. The `/` operator removes the 'base' environment wherever it occurs in the supplied state.

3.7 Summary

This section has described categories of language embedding using the μ-calculus. The calculus can be used to design embedded languages in terms of the semantics both of the language and its embedding within the host and each language definition takes the form of a triple: evaluator, loader and unloader. The design of an embedding must answer questions relative to the host and sibling languages: syntax; semantics; load and unload; safety criteria.

4 Example Application

This section provides an example of multiple embedded languages that can work together. When designing multiple embeddings we must consider the interaction of the languages and any safety criteria that prevent the languages interacting in undesirable ways. The μ-calculus approach explicitly represents components of the embeddings that make it easy to ensure safety criteria are achieved.

Consider writing web-applications that use relational database tables to store data and HTML to provide the user-interface. We will use the basic μ-calculus as the host language; it is representative of a general purpose host. Two languages are embedded within the host: SQL is used to process the database tables; HTML is used to produce the user-interface. See Section 2.1 for the language definitions.

4.1 Database Queries

The first step is to define an SQL-like language. We limit this to selecting fields from a named database table where the field values satisfy a given predicate expression:

```
type SQLExp(T) = ([String],String,Exp(T))
```

The evaluator for `SQL` requires an extra state component that maps table names to tables. We represent tables as sequences of table rows:

```
type SQLState = ([Value],Env,[Instr],SQLState,Tables) | Empty
type Tables = String->DBTable
type DBTable = [DBRow]
type DBRow = String->Value
```

The evaluator must handle the extra `SQL` constructs:

```
let evalSQL(eval)(s) =
  case s of
    (s,e,(ns,n,b):c,d,t) -> eval(t(n)/ns:[]:s,e,sel(b):c,d,t)
    ((vs:vss):s,e,sel(b):c,d,t) ->
      eval([],e[ns->vs],[b],(vss:s,e,cif(vs):sel(b):c,d,t),t)
    (true:rs:vss:s,e,cif(vs):c,d,t) -> eval(vss:(vs:rs):s,e,c,d,t)
    (false:rs:vss:s,e,cif(vs):c,d,t) -> eval(vss:rs:s,e,c,d,t)
    else evalExp(eval')(s,e,c,d)
           where (s,e,c,d,t)    = s
                 eval'(s,e,c,d) = eval(s,e,c,d,t)
  end
```

The `SQL` evaluator defined above detects `SELECT` expressions at the head of the control, looks up the table in the environment `t(n)` and selects the named fields `t(n)/ns`. The machine then uses the new instructions `sel` and `cif` to process each value-tuple in turn and build up a sequence of values that satisfy the predicate expression `b`.

4.2 Web Page Generation

The `HTML` language is defined as follows:

```
type HTML(T) = [Row(T)]
type Row(T) = Row[Col(T)] | ForRow([String],String,Row(T))
type Col(T) = Col(Exp(T)) | ForCol([String],String,Exp(T))
```

The state for the `HTML` language uses a new state component that models the output:

```
type HTMLState = ([Value],Env,[Instr],HTMLState,[String]) | Empty
```

The evaluator is defined as follows:

```
let evalHTML(eval)(s) =
  case s of
    (s,e,[rs]:c,d,o) -> eval(s,e,:rs:tend:c,d,"<TABLE>":o)
    (s,e,tend:c,d,o) -> eval(s,e,c,d,"</TABLE>":o)
    (s,e,Row(cs):c,d,o) -> eval(s,e,cs:rend:c,d,"<TR>":o)
    (s,e,rend:c,d,o) -> eval(s,e,c,d,"</TR>":o)
    (s,e,Col(b):c,d,o) -> eval(s,e,b:cend:c,d,"<TD>":o)
    (v:s,e,cend:c,d,o) -> eval(s,e,c,d,"</TD>":v:o)
```

```
    (s,e,for(ns,n,r):c,d,o) -> eval(e(n):s,e,next(ns,r):c,d,o)
    (vs:vss:s,e,next(ns,r):c,d,o) -> eval(vss:s,e[ns->vs],r:c,d,o)
    ([]:s,e,next(ns,r):c,d,o) -> eval(s,e,c,d,o)
    else evalExp(eval')(s,e,c,d)
           where (s,e,c,d,o)    = s
                 eval'(s,e,c,d) = eval(s,e,c,d,o)
end
```

The HTML evaluator defined above detects table declarations at the head of the control. HTML output is built up as each element of the table declaration is processed. The evaluator uses the new instructions tend, rend and cend to produce the terminating tags. The for-loops within rows and columns are processed using the new instructions for and next.

4.3 Language Collaboration and Safety Criteria

The state of the host language must merge the requirements of the two embedded languages. Therefore:

```
let evalExp'(eval)(s,e,c,d,t,o) = evalExp(eval')(s,e,c,d)
      where eval'(s,e,c,d) = eval(s,e,c,d,t,o)
```

The load and unload mappings for the language definitions can then perform the appropriate state projections:

```
let loadSQL((s,e,c,d,t,o),sql) = (s,e,sql:c,d,t)
let unloadSQL((s,e,c,d,t),(_,_,_,_,_,o)) = (s,e,c,d,t,o)
let loadHTML((s,e,c,d,t,o),html) = (s,e,html:c,d,o)
let unloadHTML((s,e,c,d,o),(_,_,_,_,t,_)) = (s,e,c,d,t,o)
```

The definition above specifies two languages and how they interact. Each language requires state—t for database tables and o for the HTML output. The specification requires that these states are maintained separately and interact in well-defined ways (in the calculus this is specified by scoping rules). Any implementation that is consistent with μ, SQL and HTML is required to respect this safety criteria.

5 Conclusions

In this paper we defined the μ-calculus which allows languages and homogeneous language embeddings to be specified. We showed that it is sufficiently expressive that it can be used to add new language features to itself in a coherent fashion. We identified and defined functional, non-functional, ad-hoc and uniform language embedding categories. We finally showed how the μ-calculus can be used to specify how DSLs such as an HTML generation language and SQL can be embedded within each other.

References

[1] J. Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, Aug. 1986.

[2] M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *Proc. OOPSLA'04*, Vancouver, Canada, 2004. ACM SIGPLAN.

[3] L. Cardelli, F. Matthes, and M. Abadi. Extensible grammars for language specialization. In *Proc. Fourth International Workshop on Database Programming Languages - Object Models and Languages*, pages 11–31, Aug. 1993.

[4] L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping. Technical Report 121, Digital Systems Research Centre, Feb. 1994.

[5] H. Christiansen. A survey of adaptable grammars. *SIGPLAN Notices*, 25(11):35–44, Nov. 1990.

[6] T. Clark, A. Evans, P. Sammut, and J. Willans. An executable metamodelling facility for domain specific language design. In *Proc. 4th OOPSLA Workshop on Domain-Specific Modeling*, Oct. 2004.

[7] T. Clark, P. Sammut, and J. Willans. Beyond annotations: A proposal for extensible java (xj). In *Eighth IEEE Internal Conference on Source Code Analysis and Manipulation*, pages 229–238. IEEE Computer Society, 2008.

[8] K. Czarnecki, J. O'Donnell, J. Striegnitz, and W. Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. 3016:50–71, 2004.

[9] F. Fleutot and L. Tratt. Contrasting compile-time meta-programming in Metalua and Converge. In *Workshop on Dynamic Languages and Applications*, July 2007.

[10] P. Hudak. Modular domain specific languages and tools. In *Proc. Fifth International Conference on Software Reuse*, pages 134–142, June 1998.

[11] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[12] J. R. Metzner. A graded bibliography on macro systems and extensible languages. volume 14 of *SIGPLAN Notices*, pages 57–64, Oct. 1979.

[13] S. Seefried, M. Chakravarty, and G. Keller. Optimising Embedded DSLs using Template Haskell. In *Third International Conference on Generative Programming and Component Engineering*, pages 186–205, Vancouver, Canada, 2004.

[14] T. Sheard, Z. el Abidine Benaissa, and E. Pasalic. DSL implementation using staging and monads. In *Proc. 2nd conference on Domain Specific Languages*, volume 35 of *SIGPLAN*, pages 81–94. ACM, Oct. 1999.

[15] T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Proceedings of the Haskell workshop 2002*. ACM, 2002.

[16] D. Spinellis. Reliable software implementation using domain specific languages. In G. I. Schuëller and P. Kafka, editors, *Proc. ESREL '99 — The Tenth European Conference on Safety and Reliability*, pages 627–631, Sept. 1999.

[17] L. Tratt. Domain specific language implementation via compile-time meta-programming. *TOPLAS*, 30(6):1–40, 2008.

A natural implementation of Plural Semantics in Maude [1]

Adrián Riesco and Juan Rodríguez-Hortalá

{ariesco,juanrh}@fdi.ucm.es
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain

Abstract

Recently, a new semantics for non-deterministic lazy functional(-logic) programming has been presented, in which the treatment of parameter passing was different to previous proposals like call-time choice (CRWL) and run-time choice (term rewriting). There, the semantics was formalized through the πCRWL calculus, and a program transformation to simulate πCRWL with term rewriting was proposed. In the present work we use the Maude system to implement that transformation and to get an interpreter for πCRWL, thus providing a first implementation of this new semantics. Besides, in order to improve the performance of the prototype, an implementation of the natural rewriting on-demand strategy has been developed, therefore taking the first steps towards obtaining a framework for on-demand evaluation of Maude system modules.

Key words: Language prototyping, Plural semantics, Maude, Natural rewriting, Rewriting logic.

1 Introduction

State-of-the-art implementations of functional-logic programming (*FLP*) languages (see [12] for a recent survey) use possibly non-terminating and non-confluent
constructor-based term rewrite systems (*CS's*) as programs, thus defining possibly non-strict non-deterministic functions, which are one of the most distinctive features of the paradigm [11,2]. Nevertheless, although CS's can be used as a common syntactic framework for FLP and term rewriting, the

[1] This work has been partially supported by the Spanish projects *MERIT-FORMS*-UCM (TIN2005-09207-C03-03), *DESAFIOS* (TIN2006-15660-C02-01), *PROMESAS-CAM* (S-0505/TIC/0407), and *FAST-STAMP* (TIN2008-06622-C03-01/TIN).

behavior of current implementations of these formalisms differ fundamentally, because different semantics can be assigned to a lazy functional language after introducing non-determinism. Conside-ring the program $\mathcal{P} = \{f(c(X)) \to d(X,X), X \ ? \ Y \to X, X \ ? \ Y \to Y\}$ and the expression $f(c(0) \ ? \ c(1))$, let us see what are the values for that expression under the traditional semantics for non-deterministic functions [19,13]:

- Under *call-time choice* parameter passing to compute a value for the term $f(c(0) \ ? \ c(1))$ we must first compute a (partial) value for $c(0) \ ? \ c(1)$, and then we may continue the computation with $f(c(0))$ or $f(c(1))$ which yield $d(0,0)$ or $d(1,1)$. Note that $d(0,1)$ and $d(1,0)$ are not correct values for $f(c(0) \ ? \ c(1))$ in that setting. Modern functional-logic languages like Toy [15] or Curry [12] adopt call-time choice.
  From the point of view of a denotational semantics, call-time choice parameter passing is equivalent to having a *singular semantics*, in which the substitutions used to instantiate the program rules for function application are such that the variables of the program rules range over single objects of the set of considered values.
- On the other hand, under *run-time choice* parameter passing, which corresponds to call-by-name, each argument is copied without any evaluation and so the different copies of any argument may evolve in different ways afterwards. However in $f(c(0) \ ? \ c(1))$ the evaluation of the subexpression $c(0) \ ? \ c(1)$ is needed in order to get an expression that matches the left hand side $f(c(X))$. Hence the derivations $f(c(0) \ ? \ c(1)) \to f(c(0)) \to d(0,0)$ and $f(c(0) \ ? \ c(1)) \to f(c(1)) \to d(1,1)$ are sound and compute the values $d(0,0)$ and $d(1,1)$, but neither $d(0,1)$ nor $d(1,0)$ are correct values for $f(c(0) \ ? \ c(1))$. Term rewriting is considered the standard semantics for run-time choice, and is the basis for the semantics of languages like Maude [5]. Traditionally it has been considered that run-time choice has its denotational counterpart on a *plural semantics*, in which the variables of the programs rules take their values over sets of objects of the set of considered values. But in this example we may consider the set $\{c(0), c(1)\}$ which is a subset of the set of values for $c(0) \ ? \ c(1)$ in which every element matches the argument pattern $c(X)$. Therefore, the set $\{0,1\}$ can be used for parameter passing obtaining a kind of "set expression" $d(\{0,1\},\{0,1\})$ that yields the values $d(0,0)$, $d(1,1)$, $d(0,1)$, and $d(1,0)$.

The conclusion is clear: *the traditional identification of run-time choice with a plural semantics is wrong when pattern matching is involved.* This fact was pointed out in [18] for the first time, where the $\pi CRWL$ logic was proposed as a novel formulation of a plural semantics with pattern matching. This logic shares with *CRWL* (the standard logic for call-time choice) some compositionality properties that make it more suitable than term rewriting (the standard formulation for run-time choice) for a value-based language like current implementations of FLP. For example, it is easy to see that for the previous program

the expression $f(c(0\ ?\ 1))$ has more values than the expression $f(c(0)\ ?\ c(1))$ under run-time choice, even when the only difference between them is the subexpressions $c(0\ ?\ 1)$ and $c(0)\ ?\ c(1)$, which have the same values both in call-time choice, run-time choice, and plural semantics. This is pretty incompatible with a value-based semantic view, although it can be the right choice for other kind of rewriting based languages like Maude, not limited to CS's but able to handle general term rewrite systems (*TRS's*), and in which the goal is describing the evolution of a system instead of computing values.

Maude [5] is a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications. Maude modules correspond to specifications in *rewriting logic* [16], a simple and expressive logic which allows the representation of many models of concurrent and distributed systems. This logic is an extension of equational logic; in particular, Maude *functional modules* correspond to specifications in *membership equational logic* [3], which, in addition to equations, allows the statement of *membership axioms* charac-terizing the elements of a sort. In this way, Maude makes possible the faithful speci-fication of data types (like sorted lists or search trees) whose data are defined not only by means of constructors, but also by the satisfaction of additional properties. Rewriting logic extends membership equational logic by adding rewrite rules, that represent transitions in a concurrent system. Maude *system modules* are used to define specifications in this logic. Rewriting logic is also a good semantic framework for formally specifying programming languages as rewrite theories [5, Chap. 20][17]. Moreover, since those specifications usually can be executed in Maude, they in fact become interpreters for these languages.

Exploiting the fact that rewriting logic is reflective [6], a key distinguishing feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [5, Chap. 14], a feature that makes Maude remarkably extensible and that allows many advanced metaprogramming and metalanguage applications. This powerful feature allows access to metalevel entities such as specifications or computations as usual data. In addition, the Maude system provides another module, `LOOP-MODE` [5, Chap. 17], which can be used to specify input/output interactions with the user. Thus, our program transformation, its execution, and its user interactions are implemented in Maude itself.

Although Maude provides commands to execute expressions in (metarepresented) modules, including a `metaSearch` function that performs a breadth-first search of the state space, [2] the highly non-deterministic nature of the programs obtained with the transformation avoids its use in practice. To solve this problem we have implemented the natural rewriting strategy [9], that evolves only the terms needed in the execution of an expression, avoiding to

[2] The usual Maude strategy, consisting in rewrite terms with the first possible rule is not available here because it leads to results that are not necessarily cterms, i.e., terms made only of data constructors.

$Prog$	::=	plural Name is [Rl] endp		Program
Rl	::=	$Lh \to Exp$.		Rule
Lh	::=	$f(p_1, \ldots, p_n)$	$f \in FS^n, p_i \in CTerm$	Lefthand side
Exp	::=	X	$X \in Var$	Expression
	|	$h(e_1, \ldots, e_n)$	$h \in FS^n \cup CS^n, e_i \in Exp$	

Fig. 1. Syntax

rewrite unnecessary terms. This is the first implementation of an on-demand strategy for Maude system modules,[3] and it can be considered a first stage towards on-demand execution of general rewrite theories.

This on-demand strategy has been combined with depth-first and breadth-first search, which allows to traverse the search tree in a flexible way, allowing to evaluate programs with potentially infinite branches. Furthermore, the tool also provides the option of searching with a bound in the number of rewrites, thus enhancing the performance of programs with large (possibly infinite) error branches.

The rest of the article is organized as follows: A session describing how to use the tool through examples is given in Section 2. Section 3 describes the main features of the implementation. Finally, Section 4 outlines the main characteristics of the tool and gives some future work. We refer the reader to https://gpd.sip.ucm.es/trac/gpd/wiki/PluralSemantics/Maude for the source code, examples, and further information.

2 Examples

We illustrate in this section how to use the tool by means of examples. The session is started by executing the file `plural.bin` available in the web page above. Once the tool is running we can introduce modules, that must follow the syntax shown in Figure 1 and fulfill that the rules are left-linear and their righthand sides do not use variables not present in the corresponding lefthand sides.

First, we specify the clerks example shown in [18], where we have shops with some employees, and we want to find a pair of clerks:

```
Maude> (plural CLERKS is
  branches -> madrid .
  branches -> sevilla .
  employees(madrid) -> e(john, men, clerk) .
  employees(madrid) -> e(larry, men, boss) .
  employees(vigo) -> e(mary, women, clerk) .
```

[3] On-demand strategies for Maude functional modules are described in [7].

```
  employees(vigo) -> e(james, men, boss) .
  twoclerks -> find(employees(branches)) .
  find(e(N,S,clerk)) -> p(N,N) .
endp)

Module introduced.
```

Under plural semantics, the expression `twoclerks` leads to any combination `p(name`$_1$`, name`$_2$`)`, where `name`$_i$ can be any clerk name (`john` and `mary` in the example), while run-time choice and call-time choice only lead to pairs where `name`$_1$ and `name`$_2$ coincide.

The tool reads the module and applies it the `pST` transformation, that simulates plural semantics with ordinary rewriting (see Section 3.1 for details). This transformed module can be seen with the command `(showTr .)`. We can now change the default depth-first strategy to breadth-first by using the command

```
Maude> (breadth-first .)

Breadth-first strategy selected.
```

We try now to compute the result of evaluating `twoclerks` by typing

```
Maude> (eval twoclerks .)

Result: p(john,john)
```

Since we try to find results with different names in the pair, we can ask for more answers with

```
Maude> (more .)

Result: p(john,mary)
```

Using the `more` command repeatedly we obtain all the different pairs reachable by the program until the following answer is prompted:

```
Maude> (more .)

No more results.
```

Now that we are familiar with the tool we show how to execute a more complex problem. The fearless Ulysses has been captured in his travel from Troy to Ithaca, but he knows he can persuade one of his four guardians to interchange the key for some items, that Ulysses has to obtain from the other guardians with his initial possessions:

```
(plural LAIR is
  guardians -> circe ? calypso ? aeolus ? polyphemus .
  ask(circe, trojan-gold) -> item(treasure-map) ? sirens-secret .
  ask(calypso, sirens-secret) -> item(chest-code) .
```

```
  ask(aeolus, item(M)) -> combine(M,M) .
  ask(polyphemus, combine(treasure-map, chest-code)) -> key .
```

Notice that the information given to the fourth guardian can be only obtained with our semantics, because a pair of the same variable becomes a pair of different constants. To acquire these items he uses the function `discover`, that uses the current information or tries to ask the guardians for more.

```
  discover(M) -> M ? discover(discStep(M) ? M) .
  discStep(M) -> ask(guardians, M) .
```

Finally, Ulysses escapes if he obtains the key from his initial belongings: an immeasurable amount of `trojan-gold`.

```
  escape -> open(discover(trojan-gold)) .
  open(key) -> true .
endp)
```

We use the depth-first strategy to check if the evasion is possible:

```
Maude> (depth-first .)

Depth-first strategy selected.
```

We evaluate now the term `escape` with `80` as upper bound in the number of rewrites with the command:

```
Maude> (eval [depth= 80] escape .)

Result: true
```

That is, there is a way to interchange the information in order to escape.

3 Implementation

We describe in this section the main topics of the implementation. First, we describe how the program transformation described in [18] has been accomplished. Then, we describe how to improve its execution by using the natural rewriting strategy [9].

3.1 Program transformation

The first component of our implementation is a source-to-source transformation of CS, whose adecuacy has been proved in [18]. The main idea in this transformation is postponing the pattern matching to avoid it to force an early resolution of non-determinism. To illustrate this concept, let us see the result of applying the transformation over the program $\mathcal{P}$ of Section 1:

Given a *CRWL*-program P, for every rule $(f(p_1,\ldots,p_n) \to r) \in \mathcal{P}$ its transformation is defined as:

$$pST(f(p_1,\ldots,p_n) \to r)$$

$$= \begin{cases} f(p_1,\ldots,p_n) \to r & \text{if } \rho_1 \ldots \rho_m \text{ is empty} \\ f(\tau(p_1),\ldots,\tau(p_n)) \to \textit{ if } match(Y_1,\ldots,Y_m) & \text{otherwise} \\ \qquad\qquad\qquad\qquad \textit{then } r[\overline{X_{ij}/project_{ij}(Y_i)}] & \end{cases}$$

where $\rho_1 \ldots \rho_m = p_1 \ldots p_n \mid \lambda p.(p \notin \mathcal{V} \wedge var(p) \neq \emptyset)$.
- $\forall \rho_i$, $\{X_{i1},\ldots,X_{ik_i}\} = var(\rho_i) \cap var(r)$ and $Y_i \in \mathcal{V}$ is fresh.
- $\tau : CTerm \to CTerm$ is defined by $\tau(p) = p$ if $p \in \mathcal{V} \vee var(p) = \emptyset$ and $\tau(p) = Y_i$ otherwise, for $p \equiv \rho_i$.
- $match \in FS^m$ fresh is defined by the rule $match(\rho_1,\ldots,\rho_m) \to true$.
- Each $project_{ij} \in FS^1$ is a fresh symbol defined by the single rule $projectij(\rho_i) \to X_{ij}$.

Fig. 2. pST transformation

$$\begin{aligned} \hat{\mathcal{P}} = \{\ & f(Y) \to \textit{ if } match(Y) \textit{ then } d(project(Y), project(Y)), \\ & match(c(X)) \to true, project(c(X)) \to X, \\ & \textit{if true then } X \to X, X\ ?\ Y \to X, X\ ?\ Y \to Y\} \end{aligned}$$

Now, to evaluate the expression $f(c(0)\ ?\ c(1))$, we are not forced anymore to solve the non-deterministic choice between $c(0)$ and $c(1)$, because any expression matches the variable pattern Y, therefore the step

$$f(c(0)\ ?\ c(1)) \to \textit{ if } match(c(0)\ ?\ c(1)) \textit{ then } d(project(c(0)\ ?\ c(1)), project(c(0)\ ?\ c(1)))$$

is sound. Note that the guard *if* $match(c(0)\ ?\ c(1))$ is needed to ensure that at least one of the values of the argument matches the original pattern, otherwise the soundness of the step could not be granted. Later on, after resolving the guard, different evaluation of the occurrences of $project(c(0)\ ?\ c(1))$ will lead us to the final values $d(0,0)$, $d(1,1)$, $d(0,1)$, and $d(1,0)$, which are the expected values for the expression in the original program under the plural semantics.

Program transformations like this can be easily handled in Maude thanks to its efficient use of reflection [5], that allows to manipulate (metarepresented) Maude modules (and more concretely Maude rules) as data. In [18] the transformation above is defined through the function pST shown in Figure 2, which for any program rule returns a rule to replace it, and a small set of auxiliary *match* and *project* rules for the replacement. Using the reflection features of Maude, we can implement it with an operator `pST`, that receives the rule that must be transformed and an index to create fresh function names related to this rule and returns a set of rules composed by the new rule and the as-

sociated *match* and *project* rules. If the list of ρ_i is empty, the rule is not transformated.

```
op pST : Rule Nat -> RuleSet .
ceq pST(rl T => T' [AtS] ., N) = rl T => T' [AtS] .
 if computeRhos(T) == empty .
```

If the list of ρ_i is not empty, then we must transform the rule. The `match` expression used in the *if* condition is computed with the function `createMatchExp`, that receives as argument the length of the list of ρ_i to create the same number of fresh variables. The rule that will be applied when this condition holds is defined with the operator `createMatchRule`. The operator `computeSubstitutions` calculates the project function that substitutes each ρ_i on the righthand side, and keeps the result in the table `ProjectTable`. This table is then used to make the substitution and obtain the result of the if statement. The rules associated with each projection are obtained by means of `createProjectRules`. Finally, the application of the τ function to the arguments on the lefthand side of the rule is made with `applyTau`.

```
ceq pST(rl T => T' [AtS] ., N) =
             rl applyTau(T) => 'if_then_[MatchExp, NewRHS] [AtS] .
             MatchRule ProjectRules
 if Rhos := computeRhos(T) /\ Rhos =/= empty /\
    VarsRHS := getVars(T') /\
    MatchExp := createMatchExp(size(Rhos), N) /\
    MatchRule := createMatchRule(Rhos, N) /\
    ProjectTable := computeSubstitutions(Rhos, VarsRHS, N) /\
    NewRHS := substitute(T', ProjectTable) /\
    ProjectRules := createProjectRules(Rhos, VarsRHS, N) .
```

3.2 Natural rewriting

The second component of our system is an implementation of the natural rewriting on-demand strategy [10], which became necessary to deal with the highly non-deterministic programs obtained after the transformation. This transformation is implemented by the operator `natNext` that computes the set of reachable terms by evolving the needed positions with all the possible rules.

As it is usual in other on-demand strategies, a data structure called *definitional tree* is used to encode the demand information associated to the program rules. What makes natural rewriting different and more appropriate than other on-demand strategies is that it uses a special kind of definitional

tree called *matching definitional tree* [1], that allows us to keep the pattern matching process separated from the evaluation through demanded positions. In previous strategies the encoding of these two processes were interleaved in the definitional trees, and as a consequence they lost opportunities to prune the search space. A matching definitional tree is built for each function present in the program, after a static analysis performed during the compilation. We implement this by the operator `MDTMap`, which takes the Maude representation of the transformed program and returns a map from function symbols to its corresponding definitional trees.

Once the matching definitional trees have been computed, we can use the function `mt` [9] to compute the needed positions and the rules that must be applied. Moreover, we combine this evaluation strategy with (bounded) depth-first and breadth-first search, keeping *all* the possible terms obtained from `mt` and its depth in a list that works as a stack for the depth-first strategy and as a queue for the breadth-first strategy.

Our implementation of natural rewriting is a contribution by itself, because it can be used to perform on-demand evaluation of any CS specified in Maude, not only of those used to simulate $\pi CRWL$. Nevertheless it has several limitations. First of all, although there are extensions of this strategy to deal with general TRS's [10], we have only implemented the version presented in [9], which is only able to deal with left-linear CS's. Besides, the strategy is formulated for pure CS's, and does not consider the possibility of combining the rewriting rules with eager and confluent equations, which is one of the most distinctive and useful features of Maude. Anyway the current implementation is powerful enough to help us to improve the efficiency of our simulation of $\pi CRWL$, as well as dealing with "pure" CS's specified in Maude, and we contemplate the aforementioned extensions as interesting subjects of future work.

4 Conclusions

In this work we have described a prototype implementation of the $\pi CRWL$ logic in Maude, based on a source-to-source transformation of CS. This transformation, as well as the execution of the resulting program with the natural rewriting on-demand strategy, is implemented in Maude, taking advantage of its efficient implementation of reflection, that allows to use specifications as data.

Our implementation is natural in two different ways. First of all, the natural rewriting strategy has been used to deal with the wild non-determinism explosion arising in Maude system modules. While this non-determinism is necessary to completely model check a system, because then we want to test if a given property holds in every state reachable during the evolution of the system, it is not the case when we are just computing the set of values for a given expression. In $\pi CRWL$ we only care about the final reachable values,

hence we may use an on-demand strategy like natural rewriting to prune the search space without losing interesting values.

On the other hand, both the transformation and the strategy upon which our implementation is based have been formulated at the abstraction level of source programs, that is, using the syntax and theory of TRS's, which are the foundations of Maude too. Therefore Maude was the natural choice for an object language, and as a consequence there is a small distance between the specification of the transformation and the strategy, and the code which implements them. Moreover, since the adecuacy of the transformation and the completeness and optimality of the strategy have been proved, we can affirm that the resulting implementation is correct by construction. We think that this is one of the strengths of our prototype.

Another important contribution is our implementation of the natural rewriting strategy, which has been implemented in Maude for the first time. The corresponding `natNext` operator can be used for performing on-demand evaluation of any CS specified in a Maude system module, and it is especially relevant because is the first on-demand strategy for this kind of modules, complementing the default rewrite and breadth-first search Maude commands.

As a subject of future work we contemplate the extension of natural rewriting to TRS's which are not necessarily CS's, following the theory developed in [10]. Another orthogonal extension of the strategy could go in the direction of combining non-deterministic rewriting rules evaluated on demand (from Maude system modules) with deterministic and terminating equations evaluated eagerly (from Maude functional modules). This is particularly interesting when there are fragments of a TRS which constitute a confluent and terminating TRS (like the `match` and `project` functions introduced by our transformation), that could be executed dramatically more efficiently by treating them as Maude equations.

Other extensions go in the line of adding features to the $\pi CRWL$ interpreter. Adding higher order capabilities by means of the classic transformation of [20] would be interesting and it is standard in the field of FLP. More novel would be using the matching-modulo capacities of Maude to enhance the expressivity of plural semantics, after a corresponding revision of the theory of $\pi CRWL$. Another interesting extension could come from the combination of the plural and singular semantics, using the framework for combining runtime choice with call-time choice developed in [14]. Besides, some aditional research must be done to improve the performance of the interpreter by means of some kind of sharing in the line of [4].

Last but not least, some additional effort must be invested in producing a larger collection of program examples and programming patterns that exploit the capabilities of this new semantics. In fact the development of this prototype is a step in that direction, as it allows us to do empirical experimentation with $\pi CRWL$.

Acknowledgements: The authors would like to thank Santiago Escobar for his very valuable comments and suggestions and Francisco Javier López-Fraguas and Alberto Verdejo for their useful observations of earlier versions of this paper.

References

[1] S. Antoy. Definitional trees. In H. Kirchner, G. Levi, editors, *Proceedings of the 3rd International Conference on Algebraic and Logic Programming (ALP'92)*, *LNCS* 632, pages 143–157. Springer, 1992.

[2] S. Antoy and M. Hanus. Functional logic design patterns. In Z. Hu and M. Rodríguez-Artalejo, editors, *Proceedings of the 5th International Symposium on Functional and Logic Programming (FLOPS'2002)*, *LNCS* 2441, pages 67–87. Springer, 2002.

[3] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[4] B. Braßel and F. Huch. On a tighter integration of functional and logic programming. In Z. Shao, editor, *Proceedings of the 5th ASIAN Symposium on Programming Languages and Systems (APLAS 2007)*, *LNCS* 4807, pages 122–138. Springer, 2007.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, *LNCS* 4350. Springer, 2007.

[6] M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. *Theoretical Computer Science*, 373(1-2):70–91, 2007.

[7] F. Durán, S. Escobar, and S. Lucas. On-demand evaluation for Maude. In *Proc. of 5th International Workshop on Rule-Based Programming, RULE'04*, ENTCS 124(1), pages 263–284. Elsevier, 2005.

[8] S. Eker, N. Martí-Oliet, J. Meseguer, and A. Verdejo. Deduction, strategies, and rewriting. In M. Archer, T. B. de la Tour, and C. A. Muñoz, editors, *Proceedings of the 6th International Workshop on Strategies in Automated Deduction (STRATEGIES 2006)*, *ENTCS* 174(11), pages 3–25. Elsevier, 2007.

[9] S. Escobar. Implementing natural rewriting and narrowing efficiently. In Y. Kameyama and P. Stuckey, editors, *Proceedings of the 7th International Symposium on Functional and Logic Programming (FLOPS'2004)*, *LNCS* 2998, pages 147–162. Springer, 2004.

[10] S. Escobar, J. Meseguer, and P. Thati. Natural rewriting for general term rewriting systems. In S. Etalle, editor, *Proceedings of the 14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, *LNCS* 3573, pages 101–116. Springer, 2004.

[11] J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47–87, 1999.

[12] M. Hanus. Functional logic programming: From theory to Curry. Technical report, Christian-Albrechts-Universität Kiel, 2005.

[13] H. Hussmann. *Non-Determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag, 1993.

[14] F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A flexible framework for programming with non-deterministic functions. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, 2009. To appear.

[15] F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. Rewriting Techniques and Applications (RTA'99)*, *LNCS* 1631, pages 244–247. Springer, 1999.

[16] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[17] J. Meseguer and G. Roşu. The rewriting logic semantics project. In P. Mosses and I. Ulidowski, editors, *Proceedings of the Second Workshop on Structural Operational Semantics (SOS 2005), Lisbon, Portugal, 10 July 2005*, *ENTCS* 156(1), pages 27–56. Elsevier, 2006.

[18] J. Rodríguez-Hortalá. A hierarchy of semantics for non-deterministic term rewriting systems. In *Proc. Foundations of Software Technology and Theoretical Computer Science*, 2008.

[19] H. Søndergaard and P. Sestoft. Non-determinism in functional languages. *The Computer Journal*, 35(5):514–523, 1992.

[20] D. H. Warren. Higher-order extensions to prolog: are they needed? In J. Hayes, D. Michie, and Y.-H. Pao, editors, *Machine Intelligence 10*, pages 441–454. Ellis Horwood Ltd., 1982.

GLL Parsing

Elizabeth Scott and Adrian Johnstone [1]

Department of Computer Science
Royal Holloway, University of London
Egham, Surrey, United Kingdom

Abstract

Recursive Descent (RD) parsers are popular because their control flow follows the structure of the grammar and hence they are easy to write and to debug. However, the class of grammars which admit RD parsers is very limited. Backtracking techniques may be used to extend this class, but can have explosive run-times and cannot deal with grammars with left recursion. Tomita-style RNGLR parsers are fully general but are based on LR techniques and do not have the direct relationship with the grammar that an RD parser has. We develop the fully general GLL parsing technique which is recursive descent-like, and has the property that the parse follows closely the structure of the grammar rules, but uses RNGLR-like machinery to handle non-determinism. The resulting recognisers run in worst-case cubic time and can be built even for left recursive grammars.

Key words: generalised parsing, recursive descent, RNGLR and RIGLR parsing, context free languages

Parser users tend to separate themselves into bottom-up and top-down tribes. Top-down users value the readability of recursive descent (RD) implementations of LL parsing along with the ease of semantic action incorporation. Bottom-up users value the extended parsing power of LR parsers, in particular the admissibility of left recursive grammars, although LR parsers cannot cope with *hidden* left recursion and even LR(0) parse tables can be exponential in the size of the grammar, while an LL parser is linear in the size of the grammar. Both tribes suffer from the need to coerce their grammars into forms which are deterministic, or at least near-deterministic for their chosen parsing technology. There are many examples of parser generators which extend deterministic algorithms with backtracking and lookahead[10,11,1,18,7,5], although such extensions can trap the unwary. A more formal approach to backtracking is represented by Aho and Ullman's TDPL language (recently repopularised as Parsing Expression Grammars and their associated memoized Packrat parsers). These techniques are superficially attractive because,

[1] Email:e.scott@rhul.ac.uk, a.johnstone@rhul.ac.uk

by definition, there is at most one derivation for each string in the language of a PEG, but, of course, PEG's are not context-free grammars, and as Aho and Ullman said [3, p466]

> "...it can be quite difficult to determine what language is defined by a TDPL program."

The current interest in PEG's is another manifestation of users' need for parsers which are human readable.

The Natural Language Processing (NLP) community has always had to cope with the full expressive power of context free grammars. A variety of approaches have been developed and remain popular including CYK [19], Earley [6] and Tomita style GLR parsers [15,9,13]. Although GLR parsing has not been universally adopted by the NLP community — perhaps because of its complexity compared to the easier to visualise CYK and Earley methods — GLR has the attractive property for computer science applications that it achieves linear performance on LR-deterministic grammars whilst gracefully coping with fully general grammars. Since most computing applications involve near-deterministic grammars, GLR has seen significant takeup for language re-engineering applications. It is used, for example, in ASF+SDF [16] and Stratego [17], and even Bison has a partial GLR mode [2]. We have developed [14] cubic worst-case GLR algorithms which smoothly improve their performance to linear time algorithms when processing LR grammars, but this does not address the desiderata of the top down cohort. Nobody could accuse a GLR implementation of a parser for, say, C++, of being easy to read, and by extension easy to debug.

This paper introduces a new algorithm, *Generalised LL* (GLL) parsing, which handles all (including left recursive) context free grammars; runs in worst case cubic time; runs in linear time on LL grammars and which also allows grammar rule factorisation, with consequential speed up. Most importantly, the construction is so straightforward that implementation by hand is feasible: indeed we report on the performance of a hand constructed GLL parser for ANSI C. The resulting code has the RD-property that it is essentially in one-to-one correspondence with the grammar, so parsers may be debugged by stepping through the generated code with a conventional debugger. We believe that GLL will become the generalised parsing algorithm of choice.

The insight behind GLL comes in part from our work on Aycock and Horspool style RIGLR parsers [12]. Aycock and Horspool [4] developed an approach designed to reduce the amount of stack activity in a GLR parser. Their algorithm does not admit grammars with hidden left recursion, but we have given a modified version, the RIGLR algorithm, which is general. In their original paper, Aycock and Horspool described their automata based algorithm as a faster GLR parser but it is our view that the algorithm is in closer in principle to a generalised LL parser. The RIGLR automata are

derived from the grammar rules by 'terminalising' certain instances of nonterminals in a way that removes embedded recursion. When an RIGLR traverser encounters a terminalised nonterminal, it is required to make a call to another automaton. Normally, we seek to minimise call stack activity by finding a small set of terminalisations which are complete, in the sense of eliminating all embedded recursion. However, in [12] we noted that

> if we 'terminalise' all but the topmost instance of each nonterminal, we get a parser whose stack activity mimics that of a recursive descent parser, except that left recursion is allowable!

It is this observation that lead us to apply the techniques that we developed for RNGLR and RIGLR parsing to give a general recursive descent-style algorithm. In fact we can organise the algorithm so that the parsing schedule either mimics a depth first backtracking recursive descent parser (except that recursive calls are terminated early) or so that all putative parses are synchronised with respect to reading the input. The latter synchronisation is more GLR like and causes the call stacks to be constructed in levels, and that allows a memory efficient approach to the construction of both the stacks and the associated parse trees in a full parser implementation. In this paper we focus on the former organisation.

1 The general approach

A *context free grammar* (CFG) consists of a set $\mathbf{N}$ of non-terminal symbols, a set $\mathbf{T}$ of terminal symbols, an element $S \in \mathbf{N}$ called the start symbol, and a set of grammar rules of the form $A ::= \alpha$ where $A \in \mathbf{N}$ and α is a string in $(\mathbf{T} \cup \mathbf{N})^*$. The symbol ϵ denotes the empty string. We often compose rules with the same left hand sides into a single rule using the alternation symbol, $A ::= \alpha_1 \mid \ldots \mid \alpha_t$. We refer to the strings α_j as the *alternates of* A.

A *derivation step* is an expansion $\gamma A \beta \Rightarrow \gamma\alpha\beta$ where $\gamma, \beta \in (\mathbf{T} \cup \mathbf{N})^*$ and $A ::= \alpha$ is a grammar rule. A *derivation* of τ from σ is a sequence $\sigma \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \beta_{n-1} \Rightarrow \tau$, also written $\sigma \stackrel{*}{\Rightarrow} \tau$ or, if $n > 0$, $\sigma \stackrel{+}{\Rightarrow} \tau$.

A non-terminal A is *left recursive* if there is a string μ such that $A \stackrel{+}{\Rightarrow} A\mu$.

A recursive descent parser consists of a collection of parse functions, $p_A()$, one for each non-terminal A in the grammar. The function selects an alternate, α, of the rule for A, according to the current symbol in the input string being parsed, and then calls the parse functions associated with the symbols in α. It is possible that the current input symbol will not uniquely determine the alternate to be chosen, and if A is left recursive the parse function can go into an infinite loop.

GLR parsers extend LR parsers to deal with non-determinism by spawning parallel processes, each with their own stack. This approach is made practical by combining the stacks into a Tomita-style *graph structured stack* (GSS) which recombines stacks when their associated processes converge. Direct left

recursion is not a problem for LR parsers, but hidden left recursion ($A \stackrel{*}{\Rightarrow} \beta A \mu$ where $\beta \stackrel{+}{\Rightarrow} \epsilon$) can result in non-termination.

We have used a modified type of GSS to give a Tomita-style RNGLR algorithm [13] and an Aycock and Horspool-style RIGLR algorithm [12], both of which result in parsers that can be applied to all context free grammars, including those with hidden left recursion. In the RD-based GLL algorithm, introduced in this paper, we shall use RIGLR-style 'descriptors' (see next section) to represent the multiple process configurations which result from non-determinism, and a modified GSS to explicitly manage the parse function call stacks in a way that copes with left recursion.

2 Call stacks and elementary descriptors

We begin by describing the basic approach using the grammar Γ_0

$$\begin{array}{lcl} S & ::= & A\ S\ d \mid B\ S \mid \epsilon \\ A & ::= & a \mid c \\ B & ::= & a \mid b \end{array}$$

(Note that this approach will need modification to become general, as we shall discuss in Section 3.)

A traditional recursive descent parser for Γ_0 is composed of parse functions $p_S()$, $p_A()$, $p_B()$ and a main function. The parse function contains code corresponding to each alternate, α, and these code sections are guarded by a test which checks whether the current input symbol belongs to $\textsc{first}(\alpha)$, or $\textsc{follow}(\alpha)$ if $\alpha \stackrel{*}{\Rightarrow} \epsilon$, see Section 4.1. We suppose that the input is held in a global array I of length $m+1$, and that $I[m] = \$$, the end-of-string symbol.

main() { $i := 0$
  if $(I[i] \in \{a, b, c, d, \$\})$ $p_S()$ **else** *error*()
  if $I[i] = \$$ report success **else** *error*() }

$p_S()$ { **if** $(I[i] \in \{a, c\})$ { $p_A(); p_S();$ **if** $(I[i] = d)$ { $i := i + 1$ } **else** error() }
  else { **if** $(I[i] \in \{a, b\})$ { $p_B(); p_S()$ } }

$p_A()$ { **if** $(I[i] = a)$ { $i := i + 1$ }
  else if $(I[i] = c)$ { $i := i + 1$ } **else** error() }

$p_B()$ { **if** $(I[i] = a)$ { $i := i + 1$ }
  else if $(I[i] = b)$ { $i := i + 1$ } **else** error() }

(Here *error*() is a function that terminates the algorithm and reports failure.)

Of course, Γ_0 is not LL(1) so this algorithm will not behave correctly without some additional mechanism for dealing with non-determinism. We address this by converting the function calls into explicit call stack operations using stack *push* and *goto* statements in the usual way. We also partition the body

of those functions whose corresponding nonterminal is not LL(1) and separately label each partition. In practice, then, some *goto* statements will have several target labels, corresponding to these multiple partitions: for example, this will be the case for the nonterminal S in Γ_0. We use descriptors to record each possible choice, and replace termination in the RD algorithm with execution re-start from the point recorded in the next descriptor. Instead of calls to the error function, the algorithm simply processes the next descriptor and it terminates when there are no further descriptors to be processed.

In detail, an *elementary descriptor* is a triple (L, s, j) where L is a line label, s is a stack and j is a position in the input array I. We maintain a set $\mathcal{R}$ of current descriptors. At the end of a parse function and at points of nondeterminism in the grammar we create a new descriptor using the label at the top of the current stack. When a particular execution of the algorithm stops, at input $I[i]$ say, the top element L is popped from the stack $s = [s', L]$ and (L, s', i) is added to $\mathcal{R}$ (if it has not already been added). We use $\mathcal{POP}(s, i, \mathcal{R})$ to denote this action. Then the next descriptor (L', t, j) is removed from $\mathcal{R}$ and execution starts at line L' with call stack t and input symbol $I[j]$. The overall execution terminates when the set $\mathcal{R}$ is empty. In order to allow us, later, to combine the stacks we record both the line label L and the current input buffer index k on the stack using the notation L^k. At this interim stage we treat the stack as a bracketed list, [ ] denotes the empty stack, and we assume that we have a function $\mathcal{PUSH}(s, L^k)$ which simply updates the stack s by pushing on the element L^k. In the final version of the algorithm this will be replaced by a function *create*() which builds the GSS.

$i := 0$; $\mathcal{R} := \emptyset$; $s := [L_0^0]$
L_S: **if** $(I[i] \in \{a, c\})$ add (L_{S_1}, s, i) to $\mathcal{R}$
if $(I[i] \in \{a, b\})$ add (L_{S_2}, s, i) to $\mathcal{R}$
if $(I[i] \in \{d, \$\})$ add (L_{S_3}, s, i) to $\mathcal{R}$
L_0: **if** $(\mathcal{R} \neq \emptyset)$ { remove (L, s_1, j) from $\mathcal{R}$
if ($L = L_0$ and $s_1 = [\,]$ and $j = |I|$) report success
else { $s := s_1$; $i := j$; **goto** L }
else report failure

L_{S_1}: $\mathcal{PUSH}(s, L_1^i)$; **goto** L_A
L_1: $\mathcal{PUSH}(s, L_2^i)$; **goto** L_S
L_2: **if** $(I[i] = d)$ { $i := i + 1$; $\mathcal{POP}(s, i, \mathcal{R})$ }; **goto** L_0
L_{S_2}: $\mathcal{PUSH}(s, L_3^i)$; **goto** L_B
L_3: $\mathcal{PUSH}(s, L_4^i)$; **goto** L_S
L_4: $\mathcal{POP}(s, i, \mathcal{R})$; **goto** L_0
L_{S_3}: $\mathcal{POP}(s, i, \mathcal{R})$; **goto** L_0
L_A: **if** $(I[i] = a)$ { $i := i + 1$; $\mathcal{POP}(s, i, \mathcal{R})$; **goto** L_0 }
else{ **if** $(I[i] = c)$ { $i := i + 1$; $\mathcal{POP}(s, i, \mathcal{R})$ }
goto L_0 }

L_B: **if** $(I[i] = a)$ { $i := i + 1$; $\mathcal{POP}(s, i, \mathcal{R})$; **goto** L_0 }
else{ **if** $(I[i] = b)$ { $i := i + 1$; $\mathcal{POP}(s, i, \mathcal{R})$ }
goto L_0 }

As an example we execute the above algorithm with input *aad*\$. We begin by adding $(L_{S_1}, [L_0^0], 0)$ and then $(L_{S_2}, [L_0^0], 0)$ to $\mathcal{R}$ and then go to line L_0. We remove $(L_{S_1}, [L_0^0], 0)$ from $\mathcal{R}$ and go to line L_{S_1}. The push action sets s to $[L_0^0, L_1^0]$ and we go to L_A. The pop action adds $(L_1, [L_0^0], 1)$ to $\mathcal{R}$ and then we go back to L_0. In the same way, processing $(L_{s_2}, [L_0^0], 0)$ from $\mathcal{R}$ eventually results in $(L_3, [L_0^0], 1)$ being added to $\mathcal{R}$.

$$\mathcal{R} = \{(L_1, [L_0^0], 1), (L_3, [L_0^0], 1)\}$$

Next $(L_1, [L_0^0], 1)$ is processed. At L_1 the push action sets s to $[L_0^0, L_2^1]$ and then at L_S we add $(L_{S_1}, [L_0^0, L_2^1], 1)$ and $(L_{S_2}, [L_0^0, L_2^1], 1)$ to $\mathcal{R}$. Similarly, processing $(L_3, [L_0^0], 1)$ gives

$$\mathcal{R} = \{(L_{S_1}, [L_0^0, L_2^1], 1), (L_{S_2}, [L_0^0, L_2^1], 1), (L_{S_1}, [L_0^0, L_4^1], 1), (L_{S_2}, [L_0^0, L_4^1], 1)\}$$

Processing each of these elements in turn results in

$$\mathcal{R} = \{(L_1, [L_0^0, L_2^1], 2), (L_3, [L_0^0, L_2^1], 2), (L_1, [L_0^0, L_4^1], 2), (L_3, [L_0^0, L_4^1], 2)\}$$

Then, as $I[2] = d$, processing each of these results in

$$\mathcal{R} = \{(L_{S_3}, [L_0^0, L_2^1, L_2^2], 2), (L_{S_3}, [L_0^0, L_2^1, L_4^2], 2), (L_{S_3}, [L_0^0, L_4^1, L_2^2], 2), (L_{S_3}, [L_0^0, L_4^1, L_4^2], 2)\}$$

From this set we get

$$\mathcal{R} = \{(L_2, [L_0^0, L_2^1], 2), (L_4, [L_0^0, L_2^1], 2), (L_2, [L_0^0, L_4^1], 2), (L_4, [L_0^0, L_4^1], 2)\}$$

Processing these elements gives

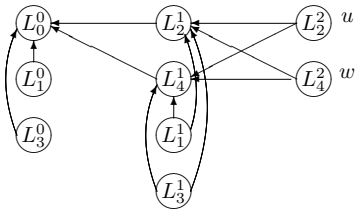
$$\mathcal{R} = \{(L_2, [L_0^0], 3), (L_2, [L_0^0], 2), (L_4, [L_0^0], 3), (L_4, [L_0^0], 2)\}$$

Since $I[3] =$ \$, processing these results in $(L_0, [\,], 3)$ and $(L_0, [\,], 2)$ being added to $\mathcal{R}$ and finally algorithm terminates and correctly reports success.

3 The GSS and the sets U_i and $\mathcal{P}$

The problem with the approach as it is described above is that for some grammars the number of descriptors created can be exponential in the size of input and the process does not work correctly for grammars with left recursion. We deal with these issues by combining the stacks into a single, global graph structure, a GSS, recording only the corresponding stack top node in the descriptor, and using loops in the GSS when left recursion is encountered. The GSS will be built by the GLL algorithm as illustrated in the modified

Γ_0-recogniser described below. The GSS combining all the stacks constructed in the example for Γ_0 previous section is



A *descriptor* is a triple (L, u, i) where L is a label, u is a GSS node and i is an integer. For example, the four elementary descriptors $\{(L_{S_3}, [L_0^0, L_2^1, L_2^2], 2)$, $(L_{S_3}, [L_0^0, L_2^1, L_4^2], 2)$, $(L_{S_3}, [L_0^0, L_4^1, L_2^2], 2)$, $(L_{S_3}, [L_0^0, L_4^1, L_4^2], 2)\}$ in the above example are replaced by two descriptors $\{(L_{S_3}, u, 2), (L_{S_3}, w, 2)\}$. As a result of this definition actions $_{\mathcal{POP}}(s, i, \mathcal{R})$ are replaced by actions which add (L, v, i) to $\mathcal{R}$ for all children v of node corresponding to the top of s.

In order to avoid creating the same descriptor twice we maintain sets $U_i = \{(L, u) \mid (L, u, i)$ has been added to $\mathcal{R}\}$. A problem arises in the case when an additional child, w, is added to u after a pop statement has been executed because the pop action needs to be applied to this child. To address this we use a set $\mathcal{P}$ which contains pairs (u, k) for which a 'pop' line has been executed. When a new child node w is added to u, for all $(u, k) \in \mathcal{P}$ if $(L_u, w) \notin U_k$ then (L_u, u, k) is added to $\mathcal{R}$, where L_u is the label of u.

These techniques are implemented via functions *add*(), *create*() and *pop*() that are formally defined in Section 4. Informally, $add(L, u, j)$ checks if there is a descriptor (L, u) in U_j and if not it adds it to U_j and $\mathcal{R}$. The function $create(L, u, j)$ creates a GSS node $v = L^j$ with child u if one does not already exist, and then returns v. If $(v, k) \in \mathcal{P}$ then $add(L, u, k)$ is called. The function $pop(u, j)$ calls $add(L_u, v, j)$ for all children v of u, and adds (u, j) to $\mathcal{P}$.

We can rewrite the algorithm from Section 2 as follows. The variable c_u holds the current GSS node, i holds the current input index and $m = |I| + 1$.

create GSS nodes $u_1 = L_0^0$, $u_0 = \$$ and an edge (u_0, u_1)
$i := 0$; $\mathcal{R} = \emptyset$; $c_u := u_1$
for $0 \leq j \leq m$ { $U_j = \emptyset$ }
L_S: **if** $(I[i] \in \{a, c\})$ $add(L_{S_1}, c_u, i)$
if $(I[i] \in \{a, b\})$ $add(L_{S_2}, c_u, i)$
if $(I[i] \in \{d, \$\})$ $add(L_{S_3}, c_u, i)$
L_0: **if** $(\mathcal{R} \neq \emptyset)$ { remove (L, u, j) from $\mathcal{R}$
$c_u := u$; $i := j$; **goto** L }
else **if** $((L_0, u_0, m) \in U_m)$ report success **else** report failure

L_{S_1}: $c_u = create(L_1, c_u, i)$; **goto** L_A
L_1: $c_u = create(L_2, c_u, i)$; **goto** L_S
L_2: **if**$(I[i] = d)$\{ $i := i + 1$; $pop(c_u, i)$ \}; **goto** L_0

L_{S_2}: $c_u = create(L_3, c_u, i)$; **goto** L_B
L_3: $c_u = create(L_4, c_u, i)$; **goto** L_S
L_4: $pop(c_u, i)$; **goto** L_0
L_{S_3}: $pop(c_u, i)$; **goto** L_0
L_A: **if** $(I[i] = a)$ { $i := i + 1$; $pop(c_u, i)$; **goto** L_0 }
else{ **if**$(I[i] = c)$){ $i := i + 1$; $pop(c_u, i)$ }; **goto** L_0 }
L_B: **if** $(I[i] = a)$ { $i := i + 1$; $pop(c_u, i)$; **goto** L_0 }
else{ **if**$(I[i] = b)$){ $i := i + 1$; $pop(c_u, i)$ }; **goto** L_0 }

Note It is not obvious how to implement the algorithm as written because few programming languages include an unrestricted goto statement that can take a non-statically visible value, which is what is implied in the **if** statement at label L_0 in the above algorithm. We discuss this in Section 5.

4 Formal definition of the GLL approach

4.1 Initial machinery

We say A is *nullable* if $A \stackrel{*}{\Rightarrow} \epsilon$. We define $\textsc{first}_{\mathbf{T}}(A) = \{t \in \mathbf{T} | \exists \alpha (A \stackrel{*}{\Rightarrow} t\alpha)\}$ and $\textsc{follow}_{\mathbf{T}}(A) = \{t \in \mathbf{T} \mid \exists \alpha, \beta (S \stackrel{*}{\Rightarrow} \alpha A t \beta)\}$. If A is nullable we define $\textsc{first}(A) = \textsc{first}_{\mathbf{T}}(A) \cup \{\epsilon\}$ and $\textsc{follow}(A) = \textsc{follow}_{\mathbf{T}}(A) \cup \{\$\}$. Otherwise we define $\textsc{first}(A) = \textsc{first}_{\mathbf{T}}(A)$ and $\textsc{follow}(A) = \textsc{follow}_{\mathbf{T}}(A)$. We say that a nonterminal A is $LL(1)$ if (i) $A ::= \alpha$, $A ::= \beta$ imply $\textsc{first}(\alpha) \cap \textsc{first}(\beta) = \emptyset$, and (ii) if $A \stackrel{*}{\Rightarrow} \epsilon$ then $\textsc{first}(A) \cap \textsc{follow}(A) = \emptyset$.

We use L_u to denote the line label corresponding to a GSS node u.

A GLL recogniser includes labelled lines of three types: return, nonterminal and alternate. Return labels, R_{X_i}, are used to label the main loop of the algorithm and what would be parse function call return lines in a recursive descent parser. Nonterminal labels, L_X, are used to label the first line of what would be the code for the parse function for X in a recursive descent parser. Alternate labels, L_{X_i}, are used to label the first line of what would be the code corresponding to the i^{th}-alternate, α_i say, of X.

The algorithm also employs three functions *add*(), *create*() and *pop*() which build the GSS and create and store processes for subsequent execution, and a function *test*() which checks the current input symbol against the current nonterminal and alternate. These functions are defined as follows.

test(x, A, α) {
if $(x \in \textsc{first}(\alpha))$ or $(\epsilon \in \textsc{first}(\alpha)$ and $x \in \textsc{follow}(A))$ { return true }
else { return false } }

add(L, u, j) { **if** $((L, u) \notin U_j$ { add (L, u) to U_j, add (L, u, j) to $\mathcal{R}$ } }

pop(u, j) { **if** $(u \neq u_0)$ { add (u, j) to $\mathcal{P}$

for each child v of u { $add(L_u, v, j)$ } } }

$create(L, u, j)$ { **if** there is not already a GSS node labelled L^j create one
let v be the GSS node labelled L^j
if there is not an edge from v to u {
create an edge from v to u
for all $((v, k) \in \mathcal{P})$ { $add(L, u, k)$ } }
return v }

4.2 Dealing with alternates

We begin by defining the part of the algorithm which is generated for an alternate α of a grammar rule for A. We name the corresponding lines of the algorithm $code(A ::= \alpha)$.

Each nonterminal instance on the right hand sides of the grammar rules is given an instance number. We write A_k to indicate the kth instance of nonterminal A. Each alternate of the grammar rule for a nonterminal is also given an instance number. We write $A ::= \alpha_k$ to indicate the kth alternate of the grammar rule for A.

For a terminal a we define

$$code(a\alpha, j, X) \;=\; \textbf{if}(I[j] = a)\ \{\ j := j + 1\ \}\ \ \textbf{else}\ \{\ \textbf{goto}\ L_0\ \}$$

For a nonterminal instance A_k we define

$$\begin{array}{rcl} code(A_k\alpha, j, X) & = & \textbf{if}(test(I[j], X, A_k\alpha)\ \{ \\ & & \quad c_u := create(R_{A_k}, c_u, j),\ \textbf{goto}\ L_A\ \} \\ & & \textbf{else}\ \{\ \textbf{goto}\ L_0\ \} \\ & & R_{A_k}: \end{array}$$

For each production $A ::= \alpha_k$ we define $code(A ::= \alpha_k, j)$ as follows. Let $\alpha_k = x_1 x_2 \ldots x_f$, where each x_l, $1 \leq l \leq f$, is either a terminal or a nonterminal instance of the form X_k.

If $f = 0$ then $\alpha_k = \epsilon$ and

$$code(A ::= \epsilon, j) \;=\; pop(c_u, j),\ \ \textbf{goto}\ L_0$$

If x_1 is a terminal then

$$\begin{array}{rcl} code(A ::= \alpha_k, j) & = & j := j + 1 \\ & & code(x_2 \ldots x_f, j, A) \\ & & code(x_3 \ldots x_f, j, A) \\ & & \ldots \\ & & code(x_f, j, A) \\ & & pop(c_u, j),\ \ \textbf{goto}\ L_0 \end{array}$$

If x_1 is a nonterminal instance X_l then

$$\begin{array}{rcl} code(A ::= \alpha_k, j) & = & \quad c_u := create(R_{X_l}, c_u, j),\ \textbf{goto}\ L_X \\ & & R_{X_l} : code(x_2 \ldots x_f, j, A) \\ & & \quad code(x_3 \ldots x_f, j, A) \\ & & \quad \ldots \\ & & \quad code(x_f, j, A) \\ & & \quad pop(c_u, j),\ \ \textbf{goto}\ L_0 \end{array}$$

4.3 Dealing with rules

Consider the grammar rule $A ::= \alpha_1 \mid \ldots \mid \alpha_t$. We define $code(A, j)$ as follows. If A is an LL(1) nonterminal then

$$\begin{array}{rcl} code(A, j) & = & \quad \textbf{if}(test(I[j], A, \alpha_1))\ \{\ \textbf{goto}\ L_{A_1}\ \} \\ & & \quad \ldots \\ & & \quad \textbf{else if}(test(I[j], A, \alpha_t))\ \{\ \textbf{goto}\ L_{A_t}\ \} \\ & & L_{A_1} : code(A ::= \alpha_1, j) \\ & & \quad \ldots \\ & & L_{A_t} : code(A ::= \alpha_t, j) \end{array}$$

If A is not an LL(1) nonterminal then

$$\begin{array}{rcl} code(A, j) & = & \quad \textbf{if}(test(I[j], A, \alpha_1))\ \{\ add(L_{A_1}, c_u, j)\ \} \\ & & \quad \ldots \\ & & \quad \textbf{if}(test(I[j], A, \alpha_t))\ \{\ add(L_{A_t}, c_u, j)\ \} \\ & & \quad \textbf{goto}\ L_0 \\ & & L_{A_1} : code(A ::= \alpha_1, j) \\ & & \quad \ldots \\ & & L_{A_t} : code(A ::= \alpha_t, j) \end{array}$$

4.4 Building a GLL recogniser for a general CFG

We suppose that the nonterminals of the grammar Γ are $A, \ldots, X$. Then the GLL recognition algorithm for Γ is given by:

m is a constant integer whose value is the length of the input
I is a constant integer array of size $m + 1$
i is an integer variable
GSS is a digraph whose nodes are labelled with elements of the form L^j
c_u is a GSS node variable
$\mathcal{P}$ is a set of GSS node and integer pairs
$\mathcal{R}$ is a set of descriptors

read the input into I and set $I[m] := \$$, $i := 0$
create GSS nodes $u_1 = L_0^0$, $u_0 = \$$ and an edge (u_0, u_1)

$c_u := u_1$, $i := 0$
for $0 \leq j \leq m$ { $U_j := \emptyset$}
$\mathcal{R} := \emptyset$, $\mathcal{P} := \emptyset$
if($I[0] \in$ first($S\$$)) { **goto** L_S } **else** { report failure }
L_0: **if** $\mathcal{R} \neq \emptyset$ {
remove a descriptor, (L, u, j) say, from $\mathcal{R}$
$c_u := u$, $i := j$, **goto** L }
else if $((L_0, u_0, m) \in U_m)$ { report success } **else** { report failure }

L_A: *code*(A, i)
...
L_X: *code*(X, i)

5 Implementation and experimental results

As we mentioned above, to implement a GLL algorithm in a standard programming language the **goto** statement in the main **for** loop can be replaced with a Hoare style case statement. We associate a unique integer, NR_{X_j} or NL_{X_j}, with each label and use that integer in the descriptors (so L becomes an integer variable). Of course, we could also substitute the appropriate lines of the algorithm in the case statements if we wished, removing the goto statements completely with the use of break statements.

Elements are only added to $\mathcal{R}$ once so the set $\mathcal{R}$ can be implemented efficiently as a stack or as a queue. As written in the algorithm $\mathcal{R}$ is a set so there is no specified order in which its elements are processed. If, as we have done, $\mathcal{R}$ is implemented as a stack then the effect will be a depth-first parse trace, modulo the fact that left recursive calls are terminated at the start of the second iteration. Thus the flow of the algorithm will be essentially that of a recursive descent parser.

On the other hand, $\mathcal{R}$ could be implemented as a set of subsets $\mathcal{R}_j$ which contain the elements of the form (L, u, j). In this case, if the elements of $\mathcal{R}_j$ are processed before any of those in R_{j+1}, $0 \leq j < m$, then the sets U_j and the GSS nodes will be constructed in corresponding order, with no elements of U_j created once $\mathcal{R}_j = \emptyset$. This can allow U_j to be deleted once $\mathcal{R}_j = \emptyset$.

To demonstrate practicality we have written GLL-recognisers for grammars for C and Pascal, for the grammar, Γ_1,

$$\begin{array}{lcl} S & ::= & C\ a \mid d \\ B & ::= & \epsilon \mid a \\ C & ::= & b \mid B\ C\ b \mid b\ b \end{array}$$

which contains hidden left recursion, and for the grammar, Γ_2,

$$S ::= b \mid S\ S \mid S\ S\ S$$

	Grammar	Input	GSS nodes	GSS edges	$\|U\|$	CPU secs
GLL	C	4,291	60,627	219,204	509,484	1.510
SRIGLR	C	4,291	44,510	78,519	180,114	1.436
GLL	C	36,827	564,164	2,042,019	4,737,207	13.750
SRIGLR	C	36,827	406,008	739,057	1,717,883	17.330
GLL	Pascal	4,425	19,728	26,264	48,827	0.140
SRIGLR	Pascal	4,425	21,086	29,369	79,885	1.770
GLL	Γ_1	$a^{20}b^{150}a$	45	67	3,330	0.010
SRIGLR	Γ_1	$a^{20}b^{150}a$	44	66	9,514	0.016
GLL	Γ_2	b^{300}	1,498	671,565	1,123,063	28.595
SRIGLR	Γ_2	b^{300}	1,496	446,117	896,718	16.550
GLL	Γ_2^*	b^{300}	1,198	357,907	583,654	8.060
SRIGLR	Γ_2^*	b^{300}	1,796	359,400	899,405	12.930

Table 1

on which standard GLR parsers are $O(n^4)$. The GLL-recognisers for C, Γ_1 and Γ_2 were written by hand, demonstrating the relative simplicity of GLL implementation. For C, the GTB tool [8] was used to generate the FIRST sets and implementation was made easier by the fact that the grammar is ϵ-free. For Pascal, the recogniser was generated by the newly created GLL-parser generator algorithm that has been added to GTB.

Of the common generalised parsers, the GLL algorithm most closely resembles the Aycock and Horspool style RIGLR algorithm, mentioned above, in which a set of automata which correspond to grammar non-terminals call each other via a common stack. The RIGLR algorithm can be tuned by selecting which non-terminal instances in the grammar generate an automaton call, trading execution time for automaton space. In the most space efficient version, which we call SRIGLR, all non-terminal instances generate a call. We have used GTB to build SRIGLR recognisers which we have compared to the corresponding GLL recognisers.

The input strings for C are a Quine-McCluskey Boolean minimiser (4,291 tokens) and the source code for GTB itself (36,827 tokens). The input string for Pascal is a program that performs elementary tree construction and visualisation (4,425 tokens). The input has already been tokenised so no lexical analysis needed to be performed. The results are shown in Table 1.

We can see that, as well as being easy to write, GLL recognisers perform well. The slower times for Γ_2 arise because the SRIGLR algorithm factors the grammar as it builds the automaton. The results for Γ_2^*

$$S \ ::= \ b \mid S \ S \ A \qquad A \ ::= \ S \mid \epsilon$$

in which the grammar is factored, demonstrate the difference. A GLL recogniser for the equivalent EBNF grammar $S ::= b \mid S\ S\ (S \mid \epsilon)$ runs in 4.20 CPU seconds on b^{300}, indicating that GLL recogniser performances can be made even better by simple grammar factorisation. This advantage is also displayed by the Pascal data; the Pacsal BNF grammar used was obtained from the EBNF original and hence is also simply factored. In general, such factorisation can be done automatically and will not change the user's view of the algorithm flow.

6 Conclusions and final remarks

We have shown that GLL recognisers are relatively easy to construct and are also practical. They have the desirable property of recursive descent parsers in that the parser structure matches the grammar structure. It is also possible to extend the GLL algorithm to EBNF grammars, allowing factorisation, and the use of iteration in place of recursion, to make the resulting parsers even more efficient.

The version of the GLL algorithm discussed here is only a recogniser: it does not produce any form of derivation. However, all the derivation paths are explored by the algorithm and it is relatively easy to modify the algorithm to produce Tomita-style SPPF representations of all the derivations of an input string. The modification is essentially the same as that made to turn an RIGLR recogniser into a parser, as described in [12].

References

[1] *JAVACC home page*. `http://javacc.dev.java.net`, 2000.

[2] *Gnu Bison home page*. `http://www.gnu.org/software/bison`, 2003.

[3] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume 1 — Parsing of *Series in Automatic Computation*. Prentice-Hall, 1972.

[4] John Aycock and Nigel Horspool. Faster generalised LR parsing. In *Compiler Construction, 8th Intnl. Conf, CC'99*, volume 1575 of *Lecture Notes in Computer Science*, pages 32 – 46. Springer-Verlag, 1999.

[5] Peter T. Breuer and Jonathan P. Bowen. A PREttier Compiler-Compiler: Generating higher-order parsers in C. *Software Practice and Experience*, 25(11):1263–1297, November 1995.

[6] J Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[7] Adrian Johnstone and Elizabeth Scott. Generalised recursive descent parsing and follow determinism. In Kai Koskimies, editor, *Proc. 7th Intnl. Conf.*

Compiler Construction (CC'98), Lecture Notes in Computer Science 1383, pages 16–30, Berlin, 1998. Springer.

[8] Adrian Johnstone and Elizabeth Scott. Proofs and pedagogy; science and systems: the Grammar Tool Box. *Science of Computer Programming*, 2007.

[9] Rahman Nozohoor-Farshi. GLR parsing for ϵ-grammars. In Masaru Tomita, editor, *Generalized LR Parsing*, pages 60–75. Kluwer Academic Publishers, The Netherlands, 1991.

[10] Terence Parr. *ANTLR home page*. `http://www.antlr.org`, Last visited: Dec 2004.

[11] Terence John Parr. *Language translation using PCCTS and C++*. Automata Publishing Company, 1996.

[12] Elizabeth Scott and Adrian Johnstone. Generalised bottom up parsers with reduced stack activity. *The Computer Journal*, 48(5):565–587, 2005.

[13] Elizabeth Scott and Adrian Johnstone. Right nulled GLR parsers. *ACM Transactions on Programming Languages and Systems*, 28(4):577–618, July 2006.

[14] Elizabeth Scott, Adrian Johnstone, and Giorgios Economopoulos. A cubic Tomita style GLR parsing algorithm. *Acta Informatica*, 44:427–461, 2007.

[15] Masaru Tomita. *Efficient parsing for natural language*. Kluwer Academic Publishers, Boston, 1986.

[16] M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[17] Eelco Visser. Program transformation with Stratego/XT: rules, strategies, tools, and systems in StrategoXT-0.9. In C.Lengauer et. al, editor, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer-Verlag, Berlin, June 2004.

[18] Albrecht Wöß, Markus Löberbauer, and Hanspeter Mössenböck. Ll(1) conflict resolution in a recursive descent compiler generator. In G.Goos, J. Hartmanis, and J. van Leeuwen, editors, *Modular languages (Joint Modular Languages Conference 2003)*, volume 2789 of *Lecture Notes in Computer Science*, pages 192–201. Springer–Verlag, 2003.

[19] D H Younger. Recognition of context-free languages in time n^3. *Inform. Control*, 10(2):189–208, February 1967.

Efficient Earley Parsing with Regular Right-hand Sides

Trevor Jim [1] Yitzhak Mandelbaum [2]

AT&T Labs Research

Abstract

We present a new variant of the Earley parsing algorithm capable of efficiently supporting context-free grammars with regular right hand-sides. We present the core state-machine driven algorithm, the translation of grammars into state machines, and the reconstruction algorithm. We also include a theoretical framework for presenting the algorithm and for evaluating optimizations. Finally, we evaluate the algorithm by testing its implementation.

Key words: Context-free grammars, Earley parsing, regular right sides, scannerless parsing, transducers, augmented transition networks

1 Introduction

This paper introduces a new parsing algorithm with three important features: it handles arbitrary context-free languages, it supports scannerless parsing, and it directly handles grammars with regular right sides.

These features all come into play when generating parsers for message formats used in network protocols such as HTTP, mail, and VoIP (SIP). These formats are defined by grammars in English-language specification documents called "Requests for Comments." The grammars are intended as documentation and therefore do not obey the restrictions typically imposed by parser generators used in the programming languages community. The grammars cannot be directly handled by existing parser generators, [3] so parsers are usually written by hand, an error-prone process that can lead to interoperability problems. Our algorithm can handle these grammars directly, without requir-

[1] Email: trevor@research.att.com
[2] Email: yitzhak@research.att.com

[3] There are a few tools which can take RFC grammars as input, but they do not generate correct parsers because of grammar ambiguities.

ing any hand-rewriting—an important consideration since the grammars are complex, ambiguous, and large, with hundreds of nonterminals.

These features are also useful in other areas, such as parsing programming languages. Many others have already pointed out the advantages of using abitrary context-free language parsers [9] and scannerless parsers [12] for programming languages, but regular right sides are less appreciated, and less supported by parser generators.

The principal advantage of regular right sides is that they are more concise and readable than their non-regular equivalents. This is demonstrated by the fact that grammars in documentation typically use regular right sides. This includes not only the Requests for Comments we have already mentioned, but also programming language documentation. For example, Python, Haskell, Ocaml, and even Pascal use some form of regular right side grammars in their official language specifications.

When languages are documented with a more restricted grammar formalism, this is often expressly so that their grammars can be used with a parser generator. The most well-known example is C, whose grammar is formulated to be compatible with YACC [6] (see Appendix B of Kernighan and Ritchie [8]). This is convenient for compiler writers, but perhaps less useful to the much larger community of C programmers.

Our parsers are based on Earley's parsing algorithm [5], so they can parse arbitrary context-free languages. They are scannerless, so they do not require syntax specifications to be split into separate lexing and parsing phases (although, lexers can be employed if desired). Finally, they handle regular right sides *directly*, without "desugaring" into a grammar without regular right sides.

Directly handling regular right sides is important for two reasons. First, desugaring introduces difficulties for the parser generator in relating the desugared grammar to the original for processing semantic actions, reporting errors, and for visualizing and debugging the parsing process. Second, and more importantly, desugaring results in less efficient parsers. Intuitively, this is because parsers must work harder when their grammars have more ambiguities. Regular expressions are highly ambiguous (e.g., $(x^*)^*$) and desugaring promotes these ambiguities into the grammar's nonterminals, that is, into parse trees. Our benchmarks show an overhead of 22–58% for desugaring.

Theoretically, our parsers are most closely related to the *augmented transition networks (ATNs)* of Woods [13], which also handle regular right sides directly, and which have previously been used for Earley parsing [2]. Our formalism uses a new kind of finite-state transducer which allows us to express some optimizations which significantly improve performance over ATNs. Some of these optimizations are similar to ones used by the Earley parsers of Aycock and Horspool [1], which do not directly handle regular right sides.

In this paper, we present and evaluate the new Earley parsing algorithm. In Section 2, we introduce a new *parsing transducer*, and present a non-

$$S = A \mid x^*y \qquad A = \epsilon \mid BAC \qquad B = x \qquad C = y$$

Fig. 1. Grammar for the language $x^n y^n \mid x^*y$; S is the start nonterminal.

deterministic parsing algorithm in which the transducer acts as the control for a pushdown automaton. We formalize the relationship between grammars and parsing transducers and specify sufficient conditions under which a given transducer will parse the language of a given grammar. In Section 3, we present an algorithm for constructing an appropriate transducer given a grammar as input. In Section 4, we present a deterministic, Earley-style, recognition algorithm based on parsing transducers and show that, for a given transducer, it recognizes the same language as the nondeterminisitic pushdown algorithm. We describe how to adapt the algorithm for parsing and present an algorithm for parse reconstruction in Section 5. We evaluate the parsing algorithm by comparing its performance to previous approaches in Section 6, and discuss related work in Section 7.

2 Foundations

Here we lay out the theoretical foundations of our parsing algorithm, omitting proofs due to space limitations.

2.1 Grammars and languages

A *grammar* G is a four-tuple $(\Sigma, \Delta, A_0, \mathcal{L})$ where

- Σ is a finite set of terminals;
- Δ is a finite set of nonterminals;
- $A_0 \in \Delta$ is the start nonterminal; and
- $\mathcal{L}$ is a family of regular languages over $(\Sigma \cup \Delta)$, indexed by Δ: $\mathcal{L}_{A_0}, \ldots$.

We use A, B, C to range over nonterminals, x, y, z to range over terminals, w to range over sequences of terminals, and W to range over sequences of terminals and nonterminals. ϵ is the empty sequence.

For every nonterminal A we define a language L_A by simultaneous induction:

$$\frac{w_0 A_1 w_1 \cdots A_n w_n \in \mathcal{L}_A, \quad w'_i \in L_{A_i} \text{ for all } i < n}{w_0 w'_1 w_1 \cdots w'_n w_n \in L_A} \quad (n \geq 0)$$

The language of G is defined to be the language of the start nonterminal A_0: $L_G = L_{A_0}$.

Figure 1 gives an example grammar for the language $x^n y^n \mid x^*y$, written as rules using regular expressions for the right-hand sides of the nonterminals.

$$\textsc{input}\ \frac{s \to_x t}{(s,\alpha) \Rightarrow_x (t,\alpha)} \qquad \textsc{call}\ \frac{s \xrightarrow{\textbf{call}} t}{(s,\alpha) \Rightarrow_\epsilon (t, ts\alpha)}$$

$$\textsc{reset}\ \frac{s \mapsto A, \quad t \to_A u}{(s,t\alpha) \Rightarrow_\epsilon (u,t\alpha)} \qquad \textsc{return}\ \frac{s \mapsto A, \quad v \to_A u}{(s,tv\alpha) \Rightarrow_\epsilon (u,\alpha)}$$

Fig. 2. Evaluation of parsing transducers

2.2 Parsing transducers

Our parsers are closely related to the augmented transition networks (ATNs) of Woods [13]. ATNs are pushdown machines whose control is provided by a set of automata, one for each nonterminal of the grammar; the automaton for a nonterminal corresponds to its regular right side.

We use a generalization of this approach, replacing the automata with *transducers*, that is, automata with outputs. Transducers in general can have outputs on transitions and on final states, but it is sufficient for our purposes here to consider outputs on final states only.

A *parsing transducer* T is an 8-tuple $(\Sigma, \Delta, Q, A_0, q_0, \to, \xrightarrow{\textbf{call}}, \mapsto)$ where

- Σ is a finite set of terminals;
- Δ is a finite set of nonterminals;
- Q is a finite set of states;
- $A_0 \in \Delta$ is the start nonterminal;
- $q_0 \in Q$ is the initial state; and
- $\to \subseteq Q \times (\Sigma \cup \Delta) \times Q$ is the transition relation;
- $\xrightarrow{\textbf{call}} \subseteq Q \times Q$ is the call relation;
- $\mapsto \subseteq Q \times \Delta$ is the output relation from final states to nonterminals.

We use q, r, s, t, u, v to range over states and α, β, γ to range over sequences of states, and we write $q \to_W r$ if $(q, W, r) \in \to$. A *configuration* is a pair (q, α) where α acts as a stack that grows to the left. Figure 2 defines $\Rightarrow_w$, a single-step evaluation relation for configurations. The multi-step evaluation relation, $\Rightarrow_w^*$, is defined as usual. Notice that the multi-step evaluation relation defines a nondeterministic algorithm for parsing inputs with the parsing transducer.

Three of the four evaluation rules are (almost) ordinary: INPUT consumes a character of input; CALL nondeterministically starts a parse of a nonterminal at the current input position, recording some state on the stack; and RETURN detects that a nonterminal has parsed correctly (as indicated by reaching a state with that nonterminal as output) and pops the stack.

There is one twist: when a new parse is started, CALL pushes *both* the caller and the callee on the stack. RETURN uses the caller to find the next state ("return address") and pops both caller and callee from the stack. RESET

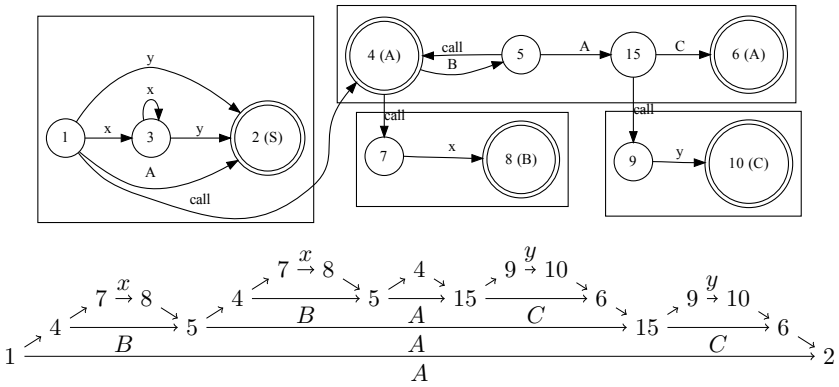


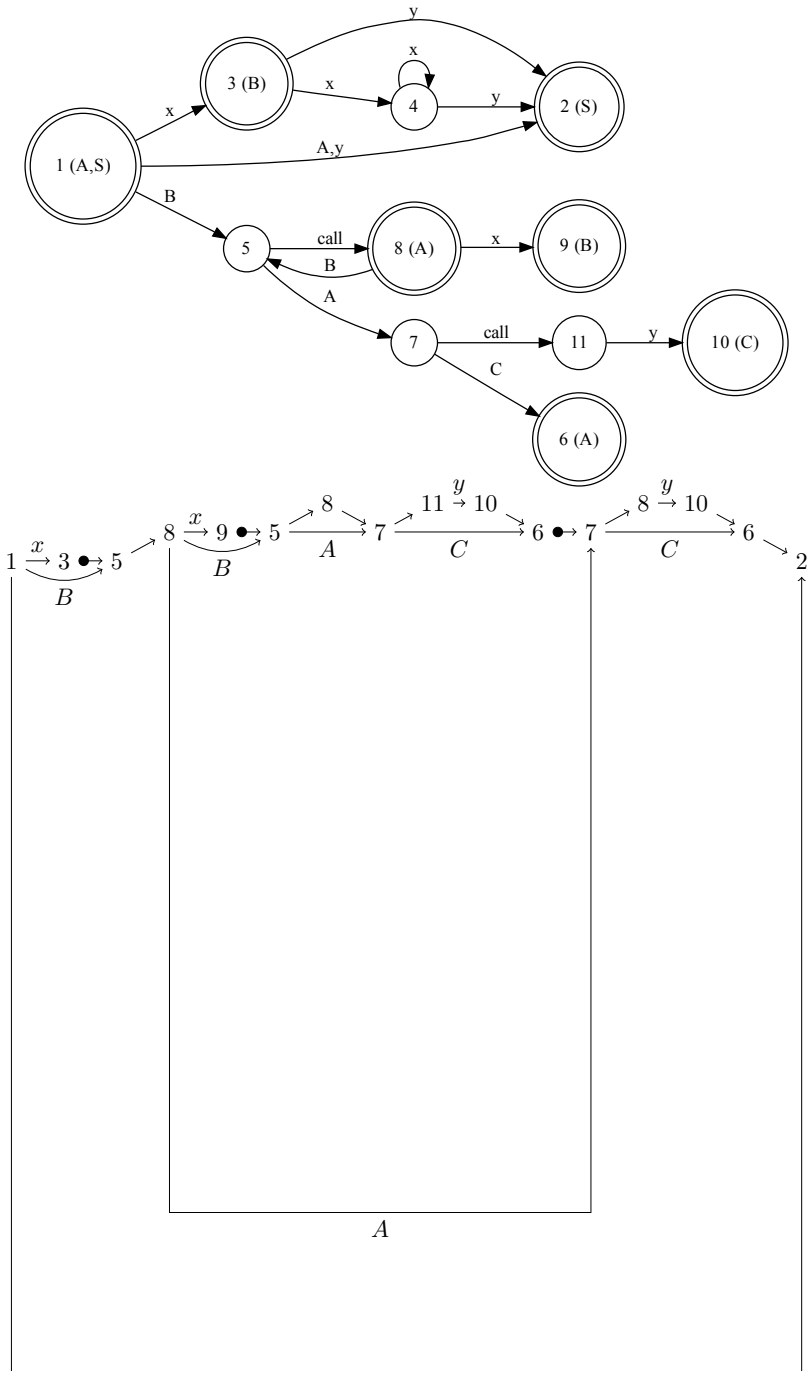
Fig. 3. A transducer for parsing $x^n y^n \mid x^* y$, and a transducer evaluation for $xxyy$.

uses the callee to find the next state, but does not pop the stack. As we will see in our discussion of Figure 4 and in Section 3, this provides more opportunities for left-factoring a grammar while building a transducer.

Finally, $L_A(q)$, *the language of* A *at* q, is $\{\, w \mid (q, q) \Rightarrow_w^* (r, q),\ r \mapsto A \,\}$, and the language of the transducer $L_T = L_{A_0}(q_0)$. Notice that we start evaluation with q_0 on the stack; the assumption is that there is always a callee at the top of the stack, for use by the rule RESET.

Figure 3 gives a transducer for parsing the language of Figure 1, and gives a transducer evaluation in graphical form. This particular transducer is roughly equivalent to the ATN formulation of Woods. Woods uses a disjoint automaton for each right-hand side, and we have boxed these in the figure: the nodes 1, 4, 7, and 9 correspond to the automata for S, A, B, and C, respectively. In the evaluation, $\nearrow$-edges indicate CALL, $\searrow$-edges indicate RETURN, arrows labeled with nonterminals indicate a parse completed by the RETURN, and arrows labeled with terminals indicate INPUT. The rule RESET is not used in this evaluation.

A more efficient transducer for the same language is given in Figure 4. In this transducer, there are no longer disjoint state machines corresponding to the right-hand sides. For example, state 8 begins a simultaneous parse of both A and B, and state 1 begins a simultaneous parse of S, A, and B (in particular, the parses of S and B have been left-factored). The evaluation makes use of the RESET rule three times, indicated by the dotted arrow. Notice that the evaluation is shorter and the stack does not grow as high as in the first evaluation. In Section 3, we describe an algorithm for constructing optimized transducers like this one.


1 (A,S)
3 (B)
4
2 (S)
5
8 (A)
9 (B)
7
11
10 (C)
6 (A)
x
y
A,y
B
call
A
C
1
2
3
5
6
7
8
9
10
11
A
B
C

2.3 Grammars and transducers

Our examples show that there can be multiple transducers capable of parsing a grammar. Here, we sketch conditions sufficient to ensure that a grammar and a transducer accept the same language. These conditions essentially say: *the transition relation $\rightarrow$ of the transducer induces the language of a right-hand side at callees*. An important consequence is that transformations like determinization and minimization which do not change the language induced by $\rightarrow$ can be used to optimize transducers.

We write $\rightarrow_W^*$ for the iterated composition of $\rightarrow$, and define $\mathcal{L}_A(q) = \{\, W \mid q \rightarrow_W^* r \wedge r \mapsto A \,\}$. We say q *is a callee* if $q = q_0$ or $\exists r.r \xrightarrow{\textbf{call}} q$. We say that a parsing transducer $T = (\Sigma, \Delta, Q, A_0, q_0, \rightarrow, \xrightarrow{\textbf{call}}, \mapsto)$ *is a transducer for* grammar $G = (\Sigma, \Delta, \mathcal{L}, A_0)$ if the following conditions hold.

(T0) $\mathcal{L}_{A_0}(q_0) = \mathcal{L}_{A_0}$.

(T1) If q is a callee and $q \rightarrow_A r$, then $\mathcal{L}_A(q) = \mathcal{L}_A$.

(T2) If $q \rightarrow_W r \rightarrow_A s$, then $r \xrightarrow{\textbf{call}} t$ for some t with $\mathcal{L}_A(t) = \mathcal{L}_A$.

(T3) If q is a callee, then $\mathcal{L}_A(q) \subseteq \mathcal{L}_A$ for all A.

Theorem 1 *If T is a transducer for G, then $L_T = L_G$.*

3 From Grammar to Transducer

There are many ways to build transducers for grammars, keeping in mind only that we need to satisfy conditions **T0**–**3**. Constructing a transducer like the one in Figure 3 that corresponds to an ATN is easy: use a standard method to convert each grammar right side into a deterministic finite automaton, mark final states as outputting their corresponding nonterminals, and add **call**-edges from each state with a nonterminal out-transition to the corresponding nonterminal's automaton.

Building an optimized transducer like the one in Figure 4 is only a little more involved. The essential idea is to ensure that in any sequence of calls, only the first is encoded using call, while the remainder are encoded as epsilon transitions. This change exposes more opportunities for left-factoring during determinization. We present the algorithm in Figure 5. For each grammar rule $A = r$, we convert the right side, r, into a nondeterministic finite automaton by $conv(r)$, resulting in a triple of states (s_A, F, E). The state s_A is the initial state for A, and F and E are final states. The construction ensures that all paths from s_A to F contain a non-**call**, non-ϵ transition, and that all paths from s_A to E contain only ϵ transitions. This invariant is used in the cases $r = (r_2 r_3)$ and $r = (r_2^*)$ to ensure condition **T2**.

The function $thompson'$ is Thompson's algorithm for converting a regular expression to an NFA [11], extended with a case for a nonterminal A: it produces both an A-transition and a **call**-transition. In contrast, the construction

$thompson'(A) =$
  Let s, F be fresh states
  Let s_A be the initial state for A
  Build $s \rightarrow_A F$
    $s \rightarrow_{\mathbf{call}} s_A$
  Return s, F

$conv(r) =$
  Let s, F, E be fresh states

A: Build $s \rightarrow_A F$
    $s \rightarrow_\epsilon s_A$
    $s \rightarrow_\epsilon E$ if A is nullable

ϵ: Build $s \rightarrow_\epsilon E$
  (F is disconnected)

x: Build $s \rightarrow_x F$
  (E is disconnected)

$r_2 r_3$:
  Let $s_2, F_2, E_2 = conv(r_2)$
  Let $s_3, F_3, E_3 = conv(r_3)$
  Let $s_{\text{call}}, F_{\text{call}} = thompson'(r_3)$
  Build $s \rightarrow_\epsilon s_2$
    $F_2 \rightarrow_\epsilon s_{\text{call}}$
    $E_2 \rightarrow_\epsilon s_3$
    $F_3, F_{\text{call}} \rightarrow_\epsilon F$
    $E_3 \rightarrow_\epsilon E$

r_2^*:
  Let $s_2, F_2, E_2 = conv(r_2)$
  Let $s_{\text{call}}, F_{\text{call}} = thompson'(r_2)$
  Build $s, E_2 \rightarrow_\epsilon s_2$
    $s, E_2 \rightarrow_\epsilon E$
    $F_2, F_{\text{call}} \rightarrow_\epsilon s_{\text{call}}$
    $F_2, F_{\text{call}} \rightarrow_\epsilon F$

$r_2 | r_3$:
  Let $s_2, F_2, E_2 = conv(r_2)$
  Let $s_3, F_3, E_3 = conv(r_3)$
  Build $s \rightarrow_\epsilon s_2, s_3$
    $F_2, F_3 \rightarrow_\epsilon F$
    $E_2, E_3 \rightarrow_\epsilon E$

Return s, F, E.

Fig. 5. Converting a regular right side to a nondeterministic automaton as part of transducer construction. Note that we only show the nonterminal case of the *thompson'* function.

$conv(A)$ uses an ϵ-transition instead of a **call**-transition, essentially performing some left-factoring. Note that *thompson'* is used in the cases $r = (r_2 r_3)$ and $r = (r_2^*)$ rather than *conv* exactly to ensure **T2**.

After we convert all right sides, we mark final states with the appropriate outputs, and apply standard determinization and minimization transformations.

4 Earley Parsing for Transducers

Earley parsing is a top-down parsing method that can parse any context-free language [5]. It performs a breadth-first search of all possible leftmost derivations of the input string. Earley parsing has an upper bound of $O(n^3)$, where n is the size of the input, and Earley showed that his algorithm works in $O(n^2)$ time for unambiguous grammars, and in linear time for a large class of grammars. Our version of Earley retains the $O(n^3)$ upper bound.

$$\textsc{init} \qquad (q_0, 0) \in E_0$$

$$\textsc{scan} \qquad \frac{(t,i) \in E_{j-1} \quad t \rightarrow_{x_j} s}{(s,i) \in E_j}$$

$$\textsc{predict} \qquad \frac{(t,i) \in E_j \quad t \xrightarrow{\textbf{call}} s}{(s,j) \in E_j}$$

$$\textsc{complete} \quad \frac{(t,k) \in E_j \quad t \mapsto A \quad (u,i) \in E_k \quad u \rightarrow_A s}{(s,i) \in E_j}$$

Fig. 6. The Earley sets

In the rest of this section we show how Earley parsing can be applied to our parsing transducers.

4.1 The Earley Sets

For an input $x_1 x_2 \ldots x_n$, we define the *Earley sets* to be the least sets satisfying rules of Figure 6. Each element $(q, i) \in E_j$ is called an *Earley item* and represents a parse which started at input position i and has progressed to position j, resulting in transducer state q. Thus, rule INIT says that we start parsing in state q_0 at input position 0, SCAN corresponds to INPUT, and PREDICT corresponds to CALL. Rule COMPLETE corresponds to both RETURN and RESET, where the return address for a parse starting at i is found from the set E_i.

Technically, the Earley set construction defines a *recognizer* that accepts an input if $(q, 0) \in E_n$ where $q \mapsto A_0$. Building a parse tree for a recognized input requires more work, as we discuss in Section 5.

Theorem 2 *If T is a transducer for G, then w is accepted by the Earley sets for T iff $w \in L_G$.*

4.2 Building the Earley Sets

The Earley sets can be built by first "seeding" E_0 by rule INIT, then processing E_0, E_1, ..., E_n in sequence as follows:

(i) For the current set E_j, apply the rules PREDICT and COMPLETE until no more items can be added.

(ii) Use SCAN to seed E_{j+1}.

It is well known that this algorithm is inefficient when the grammar has nullable nonterminals (A is nullable if $\epsilon \in L_A$). If a nullable A is called in E_j, it will also return to E_j, having consumed no terminals. This means that

$$\textsc{nncomplete}\ \frac{(t,k) \in E_j \quad t \mapsto A \quad (u,i) \in E_k \quad u \to_A s}{(s,i) \in E_j}\ (j \neq k)$$

$$\textsc{ncaller}\ \frac{(t,i) \in E_j \quad t \xrightarrow{\textbf{call}} s \mapsto A \quad t \to_A u}{(u,i) \in E_j}$$

$$\textsc{ncallee}\ \frac{(t,i) \in E_j \quad t \xrightarrow{\textbf{call}} s \mapsto A \quad s \to_A u}{(u,j) \in E_j}$$

$$\textsc{ninit}\ \frac{q_0 \mapsto A \quad q_0 \to_A r}{(r,0) \in E_0}$$

Fig. 7. Rules for handling nullable symbols

COMPLETE will be invoked with $j = k$. Unfortunately, COMPLETE needs to consider the *full* set E_k to look for A-transitions, and if $j = k$ then E_k will in fact be the set under construction. Nullable nonterminals thus require extra processing.

For this reason, our algorithm replaces the rule COMPLETE with the rules in Figure 7. The rule NNCOMPLETE is just the special case of COMPLETE for a non-null A. The remaining rules handle null completion. NCALLER combines PREDICT with an immediate COMPLETE to the caller, and NCALLEE combines PREDICT with an immediate COMPLETE to the callee. NINIT does null completion for the initial Earley item, $(q_0, 0) \in E_0$; recall that q_0 is a special case of a callee for our parsing transducers, and so NINIT is a special case of NCALLEE.

The new rules are equivalent to the original rules provided the transducer satisfies **T2** and the following condition:

(P1) If t is a callee for A (*i.e.*, $\mathcal{L}_A(t) = \mathcal{L}_A$) and A is nullable, then $t \mapsto A$.

(Note that **P1** fails in general for our construction in Section 3, but modifying the construction to satisfy **P1** requires little work.)

To see that the rules are equivalent, consider that when any (t, i) with $t \to_A$ for nullable A is added to E_j by scanning or completion, **T2** says there will be a **call** from t, and **P1** ensures that NCALLER will perform the null completion on the transition $t \to_A$. When (t, i) is added by prediction, **P1** ensures that NCALLEE applies. The remaining case is for the initial item $(q_0, 0) \in E_0$, and here **P1** ensures that NINIT applies.

For efficiency, our algorithm applies all of the rules PREDICT, NCALLER, and NCALLEE together whenever considering an item $(t, i) \in E_j$ such that $t \xrightarrow{\textbf{call}} s$ in the closure phase.

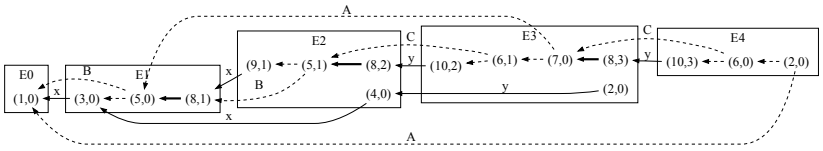


Fig. 8. The Earley graph for parsing $xxyy$.

5 Parse Reconstruction

The algorithm of Section 4 is a recognizer: it accepts exactly the inputs that parse according to the grammar, but it does not return a parse tree or parse forest for the input. Here we extend the algorithm so that trees and forests can be reconstructed for an input.

The main idea is to modify the recognizer to remember some information as it applies rules like SCAN. The stored information takes the form of a graph whose nodes are labeled with Earley items, and whose edges indicate why a node is added to an Earley set. The resulting graph embodies a parse forest, and we show how to extract parse trees from it.

Our method can be seen as an extension of the method of Aycock and Horspool [1] to handle regular right sides. It is also similar to the work of Scott [10], who showed how to efficiently modify the traditional Earley algorithm to construct a *shared packed parse forest* (SPPF). We discuss forest reconstruction in Section 5.3.

5.1 Earley graphs

The first step in reconstruction comes during the parsing process itself. We extend the algorithm of Section 4 to construct an *(Earley) parse graph* during parsing. Each node of the graph corresponds to an Earley item in an Earley set. Directed edges from a node describe how rules were used during parsing to add the Earley item to the Earley set.

We explain Earley graphs by example. Figure 8 shows the Earley graph of the transducer of Figure 4 evaluated on the input $xxyy$. There are several kinds of edges in the graph.

- Edges labeled with terminals correspond to use of the rule SCAN. For example, the item (3,0) is added to E_1 by scanning x starting from item (1,0) in E_0.
- Bold edges correspond to the use of prediction. For example, the item (8,1) is added to E_1 by prediction starting at item (5,0) in E_1.
- Dashed edges correspond to completion. The dashed edges come in pairs—they are multiedges. One edge is labeled with the nonterminal that has been completed, and it points at the node which initiated the parse of the

```
fun mk_parse_tree sym item_complete =
  fun complete_children item children =
    if empty (links item) then
      children
    else
    let l = CHOOSE from links item in
    if is_nonterminal l then
      let child = mk_parse_tree (sym_sub l) (item_sub l)  in
      complete_children (item_pred l) (child::children)
    else
      complete_children (item_pred l) children
  in
  let children = complete_children item_complete [] in
  ( sym, item_complete.i, item_complete.j, children )
```

Fig. 9. Tree Reconstruction

nonterminal. The other edge points to the node which completed the parse of the nonterminal. For example, the item (5,0) is added to E_1 as a result of completing a parse of B that was initiated by item (1,0) in E_0, and completed by item (3,0) in E_1.

The second kind of edge, for prediction, is not used by Aycock and Horspool or by Scott. It is necessary in our case to handle a subtle detail that we will discuss in Section 5.3.

Earley graphs encode sets of parse trees. For example, in the graph of Figure 8, two parse trees for S are evident. One is rooted at item (2,0) in E_3, and it indicates a complete parse of the prefix xxy by three scans. The second tree is rooted at item (2,0) in E_4, and it corresponds to the parse tree for the full input given in Figure 4.

5.2 Reconstructing a parse tree

In Figure 9, we sketch the algorithm for nondeterministically reconstructing a parse tree for the input from the Earley graph. The algorithm is defined in terms of two mutually recursive functions, `mk_parse_tree` and `complete_children`. The function `mk_parse_tree` takes two arguments: the nonterminal for which to reconstruct the parse tree (`sym`); and the node at which the nonterminal was completed (`item_complete`). It returns a parse tree consisting of the nonterminal, its span in the input, and its children (if any). Function `complete_children` takes a node and a list of children and extends the list to be a complete reconstruction of the nonterminal associated with the argument node.

`mk_parse_tree` reconstructs the children of the parse tree from right to left. It starts with an empty list and the rightmost item of the completed nonterminal (`item_complete`). It then calls `complete_children` to (recursively) traverse predecessor links until reaching an item with no links, which must be

a callee of some call. While traversing links, `complete_children` reconstructs subtrees when nonterminal links are encountered. Subtree reconstruction is accomplished with a recursive call to `mk_parse_tree`. Some items may have been the target of more than one transition, resulting in multiple links. In this case, the algorithm nondeterministically chooses a single link to follow with the abstract `CHOOSE` function.

5.3 Reconstructing a parse forest

We have argued that the Earley graph embodies all of the parse trees for an input, and have just described how to reconstruct a parse tree from the Earley graph. However, we may wish to have a more accessible representation of the parse trees than the Earley graph: a parse forest. We have implemented a parse forest reconstruction algorithm, but do not have space to describe it here in full. Instead, we will discuss some subtleties that arise due to our use of parsing transducers.

Our transducer construction is designed to reduce the number of items within Earley sets, by combining states (*i.e.*, LR(0) items) which always appear together. However, such combinations can potentially lead to a single Earley state becoming part of multiple transducer states. In particular, this can occur for callee states. For example, consider the following grammar:

$$S = x(A \mid yB) \qquad A = y(B \mid C) \qquad B = www \qquad C = z$$

The start state of B will appear in two different transducer states—one with the start state of C and one without. As is common in subset construction, multiple paths of this sort can converge into a single path within the automaton. Correspondingly, there can be multiple paths within the Earley sets which converge to a single path. Usually, convergence indicates an ambiguity at the point of convergence. However, in this case it does not (it does, however, indicate an ambiguity at some higher level in the call stack). Therefore, when reconstructing the parse forest, we must take care not treat such convergences as indicative of multiple parse paths. Note that Scott's algorithm for reconstructing parse forests does not encounter this problem because it was designed for the traditional Earley algorithm, in which start LR(0) items are never grouped together.

In order to filter such erroneous parse trees, we need to filter paths by their call source. Therefore, during parsing, we must track the set of callers for each state. Then, during reconstruction we provide the reconstruction function with the item responsible for calling the nonterminal under reconstruction. Next, we filter links whose caller sets do not include the specified caller[4] Finally, we specify the appropriate caller to the reconstruction function by providing the predecessor item of the nonterminal link.

[4] In order to filter links in this way, the caller sets of each link must be included in the Earley graph. In our implementation, we only include caller sets for callees and we filter paths only upon completion (by rejecting paths with mismatched callers).

Grammar	Input Size	Execution Time			Overhead	
		Regular	Desugared	ATN	Desugared	ATN
IMAP	330 KB	1.09	1.72	1.61	58%	48%
OCaml	228 KB	1.15	1.40	5.77	22%	400%

Table 1
Average parse times and overheads. Each average was derived from 10 trials.

6 Evaluation

We have implemented several versions of our algorithm and compare them in Table 1. "Regular" is the recognition algorithm as described in this paper. "Desugared" is the same algorithm, but using a grammar obtained desugaring regular right sides from the original grammar. We believe that this should approximate the PEP algorithm of Aycock and Horspool and show the performance advantage of directly handling regular right sides. Finally, "ATN" is a version of the algorithm that uses a transducer that closely corresponds to Woods' augmented transition networks (see the first paragraph of Section 3). This should show the performance advantage of using our optimized parsing transducers.

We tested our algorithm on two grammars, whose styles are markedly different: IMAP, the popular mail protocol [4]; and the OCaml programming language. The IMAP grammar was extracted automatically from several RFCs, and it is written in a scannerless style. We adapted the OCaml grammar from the Lex and Yacc specifications available in the OCaml compiler source distribution (3.09.3). The IMAP input is a 330KB trace generated by concatenating a smaller trace (3.3KB) multiple times. The OCaml input is ∼50 OCaml source files, totaling 228KB; the largest file is 34KB. We ran all of the experiments on MacBook Pro with a 2.33 GHz Intel Core 2 Duo processor and 2 GB of RAM.

Table 1 gives our results. The first three columns of numbers give the execution times of the three algorithms. The final two columns give the overhead of the the Desugared and ATN variants, relative to the Regular variant. Note that for both grammars, the other methods show overheads of 22–400%, demonstrating a clear advantage for our algorithm. Second, the relationships between the algorithms differs for IMAP and OCaml, which is not surprising considering the difference in the grammar styles.

7 Related Work

Earley's original algorithm was one of the first parsers for general context-free languages [5]. It does not directly handle regular right sides.

Woods' augmented transition networks [13] are a particularly elegant parsing formalism, which seems little-known in the programming language com-

munity. They directly handle regular right sides, and have been used in Earley parsing [2]. Woods also considered ATNs based on transducers [14], though not in the context of Earley parsing. Our transducers are designed specifically to be more efficient than ATNs for Earley parsing.

Aycock and Horspool's Earley parsing algorithm [1] uses automata similar to the transducers we construct in Section 3 (though they arrive there in a very different way); in particular, they satisfy a property like our **T2**. They handle nullable symbols through a grammar transformation. Their parser is not scannerless and does not directly handle regular right sides.

We are not aware of another parsing algorithm that handles general context-free grammars with regular right sides without desugaring. There has been some work on regular right sides for LR(1) subsets [7,3].

Scott shows how to construct SPPFs using Earley parsing [10]. Her algorithm is designed for the traditional Earley algorithm; it would be interesting to study how to use her binarised SPPFs for grammars with regular right sides.

References

[1] John Aycock and R. Nigel Horspool. Practical Earley parsing. *Computer Journal*, 45(6):620–630, 2002.

[2] S. M. Chou and K. S. Fu. Transition networks for pattern recognition. Technical Report TR–EE–75–39, School for Electrical Engineering, Purdue University, West Lafayette, IN, 1975.

[3] Compiler Resources, Inc. Yacc++ and the language objects library. `http://world.std.com/~compres/`.

[4] M. Crispin. Internet message access protocol — version 4rev1. `http://www.ietf.org/rfc/rfc3501.txt`, March 2003.

[5] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[6] S. C. Johnson. Yacc: Yet another compiler compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[7] Sönke Kannapinn. *Reconstructing LR Theory to Eliminate Redundance, with an Application to the Construction of ELR Parsers.* PhD thesis, Technical University of Berlin, 2001.

[8] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall, Inc., 1988.

[9] Scott McPeak and George C. Necula. Elkhound: A fast, practical GLR parser generator. In *CC 2004: Compiler Construction, 13th International Conference*, 2004.

[10] Elizabeth Scott. SPPF-style parsing from Earley recognisers. In *Proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007)*, volume 203 of *Electronic Notes in Theoretical Computer Science*, pages 53–67. Elsevier, 2008.

[11] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[12] M.G.D. van den Brand, J. Scheerder, J.J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In *Compiler Construction 2002*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158, April 2002.

[13] W. A. Woods. Transition network grammars for natural language analysis. *Communications of the ACM*, 13(10):591–606, 1970.

[14] W. A. Woods. Cascaded ATN grammars. *American Journal of Computational Linguistics*, 6(1):1–12, 1980.

TOOL PAPER: ScalaBison Recursive Ascent-Descent Parser Generator [1]

John Boyland [2] Daniel Spiewak [2]

Department of EE & Computer Science
University of Wisconsin-Milwaukee
Milwaukee, Wisconsin, USA

Abstract

ScalaBison is a parser generator accepting bison syntax and generating a parser in Scala. The generated parser uses the idea of "recursive ascent-descent parsing," that is, directly encoded generalized left-corner parsing. Of interest is that fact that the parser is generated from the LALR(1) tables created by bison, thus enabling extensions such as precedence to be handled implicitly.

1 Introduction

Recursive ascent-descent parsing was proposed by Horspool [3]. The idea is to combine the power of LR parsing with the small table size and ease of inserting semantic actions available in LL parsing. Furthermore, the generated parsers can be *directly encoded*, in that the control is handled through executable code rather than indirectly through a table that is then interpreted. In this section we describe these concepts in greater detail.

1.1 Left-corner parsing

Demers [2] introduced "generalized left corner parsing" which (roughly) combines the benefits of LL and LR parsing techniques. When using top-down or predictive parsing (LL) to parse the yield for a given nonterminal, one requires that the parser identify ("predict") which production will be used.

[1] Work supported in part by the National Science Foundation (CCF-0702635). The opinions expressed here are not necessarily those of the National Science Foundation or the US Government.

[2] Email: {boyland,dspiewak}@uwm.edu

Left-recursive grammars cannot be used in general because unbounded lookahead may be required to determine which production should be chosen. On the other hand, bottom-up (LR) parsers can follow multiple productions as long as the ambiguity is resolved by the time we reach the end of any production. Intuitively, LL parsers require that a decision be made at the start of the productions, whereas LR can wait until the end. Thus, LR theory permits a greater number of grammars to be parsed deterministically.

The greater power of LR is offset by the greater complexity of the parser, by the larger tables generated and by the limitation of where semantic actions can occur in the grammar. The last item is somewhat of a red herring, because modern parser generators such as bison based on LR(1) (or its simplification LALR(1)) permit semantic actions to appear just about anywhere an LL(1) parser generator would, assuming the grammar is indeed LL(1). [3] The larger size of tables is less of a problem for today's computers, especially when compression techniques are used. However, the greater complexity of the parser means it is much harder for the user of the parser generator to understand what is happening.

Modern LL parser generators overcome some of the limitations of LL parsing by permitting the grammar writer to include code to help disambiguate cases. This is possible because the top-down parsing technique is intuitive. The disadvantage is that the grammar starts to accrete implementation details that obscure its clarity. On the contrary, bison, especially with its precedence extensions, enables grammars to be written in a clean and declarative style.

The intuition behind generalized left-corner parsing is that during LR parsing, few productions must wait until the end to resolve ambiguity. Frequently, the production that will be used is identified long before its end. Thus in left-corner parsing, the parser switches from bottom-up to top-down parsing as soon as the correct production is identified. This has two benefits over straight LR parsing: the tables are smaller and prediction makes it easier to generate useful error messages—if one terminal is predicted, it is easy to indicate the expectation in an error message in case a different terminal is encountered.

The key technique in order to perform left-corner parsing is to determine the *recognition points* for each production in the grammar, the points where ambiguity is resolved. Horspool generalizes recognition points into *free positions* which are points where a semantic action can be inserted. The recognition point is always a free position, but not vice versa since in some unusual cases [3], non-free positions occur *after* an earlier free position. In this paper, we choose the earliest free position that has no following non-free positions as the recognition point.

[3] "Just about" because there are a few unusual cases where this correspondence does not hold.

1.2 Recursive ascent-descent parsing

Recursive *descent* parsing is a well-known implementation technique for predictive parsing. The parser is directly encoded as a set of mutually recursive functions each of which parses a particular nonterminal.

Recursive *ascent* parsing uses recursive functions to directly encode a bottom-up parser. The set of mutually recursive functions consists of one function for each LR parsing state. Pennello [7] gives an assembly language implementation of a directly encoded LR parser. It seems the concept was later invented independently by Roberts [9] and by Kruseman Aretz [4]. Direct encoding can lead to a faster parsing for the same reason that compilation usually leads to faster execution than interpretation. Horspool [3] explains that recursive ascent parsing has not been seen as practical because of the large code size (large tables) and unintuitiveness of the technique. Recursive ascent parsers would be too tedious to write by hand, and the generated parsers are not hospitable to human injection of semantic routines.

Generalized left-corner parsing's advantages *vis-a-vis* LR parsing are directly relevant: they lead to smaller tables and after disambiguation, using top-down parsing in which it is easy to place semantic actions. Horspool [3] showed that the advantages are real—parsing can be almost three times faster as opposed to with yacc, and still enable hand-editing of semantic actions.

1.3 Precedence and other extensions

The bison tool (and its forerunner yacc) includes the ability to declare the precedence and associativity of terminals enabling grammars with operators to have smaller tables. The technique gives a way to resolve shift-reduce and reduce-conflicts without the need to add new states. Any remaining parse table conflicts are resolved in a repeatable way. (Neither kind of resolution is always benign—the resulting parser may reject strings that can be generated by the grammar.) Finally, bison includes an `error` symbol that affects error recovery.

Together these extensions change the theoretical nature of the parsing problem. Thus any tool which seeks to duplicate bison's semantics of parsing cannot simply use generalized left-corner parsing theory.

2 Architecture of ScalaBison

The key design decision behind ScalaBison was to delegate the table construction to bison. This enables us to match the syntax and semantics of bison (including its parse table disambiguation techniques) without needing to duplicate the functionality. On the other hand, this decision is limiting in that we cannot create new parsing states arbitrarily – we can only reuse

(and adapt!) the ones given to us by bison. Furthermore, it also means our tool is tied to a particular textual representation of parse tables. Fortunately, the format of bison's "output" seems stable. We have been able to use bison version 1.875 as well as 2.3.

ScalaBison performs the following tasks:

(i) Invoke the bison parser generator;

(ii) Read in the grammar and generated LALR(1) tables from bison;

(iii) Determine a recognition point for each production;

(iv) Identify the set of *unambiguous nonterminals*: non-terminals occurring after the recognition point of some production;

(v) For every unambiguous nonterminal, identify a bison state to adapt into a left-corner (LC) state, and perform the adaptation;

(vi) Write out the parser boilerplate;

(vii) Write a function for each terminal (match or error) and unambiguous nonterminal (start a recursive ascent parse at its LC state);

(viii) Write a function for parsing each production after its recognition point using the previous functions for each symbol;

(ix) Write a recursive ascent function for each LC state.

In this paper, we pass over most of these tasks without comment. The interesting steps are Step iii and Step v. We briefly note however that the start symbol S will always be in the set of unambiguous nonterminals determined in Step iv because of the artificial production $S' \rightarrow S\$$ added by the generator.

2.1 Recognition Points

The recognition point for a production is determined by finding the left-most position in each production which is free and for which all following positions are free. Recall that a "free" position is one in which a semantic action can be inserted without introducing a parse conflict. At worst, the recognition point is after the end of the production.

We modify Algorithm 2 of Purdom and Brown [8] to determine free positions. The published algorithm does a computation over a graph for each state and each potential lookahead checking whether each relevant item dominates the action for the given lookahead symbol. We instead use a single graph for each LALR(1) state. We check for each item whether, in this graph, it dominates each parse action it can reach. If it doesn't, this means that at the point where the parse has progressed to this item, there is still ambiguity, and thus the item is not free.

Precedence and associativity declarations are used by bison to resolve certain shift-reduce conflicts in favor of reductions (rather than shifts). So-called

"non-associativity" declarations can even introduce parse errors. Thus with appropriate precedence declarations

```
a - b - c
```

is parsed as `(a-b)-c` and

```
e == f == g
```

is a parse error. Normally, the recognition point for binary operators is directly before the operator, but then the recursive descent part of the parser would need to be context-sensitive so that the expression starting with `b` terminates immediately rather than extending incorrectly through "`- c`" as it would normally. Thus for correctness, we force the recognition point of any production using precedence to be put at the end. This safety measure is required whenever a shift-reduce conflict is resolved in favor of a reduce (which in bison only happens with precedence).

2.2 Generating LC States

An LC parser uses LR parsing techniques until it has determined which production to use, as determined when it reaches the recognition point. At this point, the production is "announced" and the remainder of the production is parsed using LL techniques. At latest, a production is announced at the point where the LR parser would reduce it (at the end). Thus an LC parser has no reduce actions but rather "announce" actions.

Once a production is announced, the parser predicts each remaining symbol in turn. Terminals are handled by simply checking that the next input symbol matches. Nonterminals are handled by calling a routine specially generated to parse this nonterminal. Here, we revert back to LR-style parsing, and thus we need a parse state to parse this nonterminal. Following Horspool, we generate this parse state for parsing N around the core consisting of a single item $N^\sharp \rightarrow \ \vdash \ \cdot N$ for a new artificial nonterminal $N^\sharp$. The $\vdash$ is used to ensure that the item will be be seen as "core."

A similar artificial nonterminal and item is used in LR parser generation for the start state (alone); when the end of this production is reached, the parser considers an "accept" action. For an LC parser, "accept" actions are possible for any (unambiguous) nonterminal.

In order to avoid having to determine the parse actions ourselves, we find an existing LALR state that contains the item $N_0 \rightarrow \alpha \cdot N\beta$. We then adapt the LALR state's actions (see below) to get the LC state's actions. In the process of creating an LC state, we may need to create new states to receive shift/goto actions. This process continues until no new LC states must be created. Then we move on to the next nonterminal that needs its own parse state (each "unambiguous" nonterminal needs one) until all are handled.

```
 7 class_decl: CLASS TYPEID formals superclass '{' feature_list . '}'
12 feature_list: feature_list . feature ';'
13             | feature_list . error ';'
14             | feature_list . NATIVE ';'
15             | feature_list . '{' block '}'
16 feature: . opt_override DEF OBJECTID formals ':' TYPEID '=' expr
17        | . opt_override DEF OBJECTID formals ':' TYPEID NATIVE
18        | . VAR OBJECTID ':' TYPEID '=' expr
19        | . VAR OBJECTID ':' NATIVE
20 opt_override: . OVERRIDE
21             | . /* empty */

error     shift, and go to state 49
NATIVE    shift, and go to state 50
OVERRIDE  shift, and go to state 51
VAR       shift, and go to state 52
'{'       shift, and go to state 53
'}'       shift, and go to state 54

DEF  reduce using rule 21 (opt_override)

feature       go to state 55
opt_override  go to state 56
```

Fig. 1. An (augmented) LALR state generated by bison.

```
   feature_list# : |- feature_list .
12 feature_list: feature_list . feature ';'
13             | feature_list . error ';'
14             | feature_list . NATIVE ';'
15             | feature_list . '{' block '}'

error         go to state 14
NATIVE        announce rule 14
OVERRIDE      announce rule 12
VAR           announce rule 12
'{'           announce rule 15
'}'           accept feature_list
DEF           announce rule 12
$default      accept feature_list
```

Fig. 2. The LC state formed by adapting the LALR state in Fig. 1.

Figure 1 shows an example LALR state whose actions are adapted for the LC state shown in Fig. 2. For the LC state, we start with the five items shown.

Then we "close" the new LC state, by adding new items $N' \to \cdot\alpha$ for every production $N' \to \alpha$ whenever the LC includes an item with N' immediately after the dot, *provided* that the recognition point occurs after the dot in the item. This last condition distinguishes the process from traditional LR state generation. In Fig. 2, no new items are added because the items for rules (productions) 12, 14 and 15 are all at their recognition point, and the item for rule 13 has the artificial nonterminal "error" after the dot.

Shift actions lead to (potentially) new LC states after moving the dot over the appropriate symbol, again *provided* that this does not move the dot past the recognition point. Otherwise, for a shift action, we need to determine what "announce" action is appropriate at this point (see below). In Fig. 2, the only shift/goto to remain is the one for `error`.

When adapting a reduce action from the LALR state, we also need to determine what announce action is appropriate—it is not always the one the LALR state was going to reduce, because the corresponding item might not be in the LC state. Thus, both for shift actions that go past the recognition point (such as on `OVERRIDE`) and for reduce actions (such as on `DEF`), we need to determine whether an announce action should be done instead. We do this by tracing the item corresponding to the action back to find how it was added to the LALR state.

For `OVERRIDE`, we trace the shift action to the item on rule 20 which came about during closure of the LALR state from items for rules 16 and 17 which in turn came from closure on the item for rule 12. This last item is in the LC state and thus we use the "announce rule 12" action for this input. The shift action on `VAR` gives the same conclusion. The shift actions on `NATIVE` and `'{'` are mapped to "announce rule 14" and "announce rule 15" actions respectively, through simpler applications of this process. The shift action for `'}'` leads to a different outcome. When we trace it back we get to the item for rule 7, which is a core item of LALR state, but absent in the LC state. Thus no announce action is generated for `'}'`. We return to this case below. For the reduce action on `DEF`, we trace the action back to rules 16 and 17, and thus back to rule 12 and thus generate the "announce rule 12" action for `DEF`.

If the LC state contains the artificial item $N^\sharp \to \vdash N\cdot$ (as in the example, where N is `feature_list`), then we add the default action to "accept" the nonterminal N. This default action is also used for any actions left undefined previously (as with the action for `'}'`).

Although this adaptation requires some work, by using bison's LALR states, we preserve bison's resolution semantics, while avoiding the need to propagate lookaheads or to negotiate parsing conflicts using precedence rules or other policies.

Figure 3 shows the generated Scala code for the LC state in Fig. 2. The `try` block is used to handle parse errors (because the state can handle the

```
private def yystate13(yyarg1: Features) : Int = {
  var yygoto : Int = 0;
  try {
  yycur match {
    case YYCHAR('}') => yygoto = 2; yynt = YYNTfeature_list(yyarg1);
    case NATIVE() => yygoto = 1; yynt = YYNTfeature_list(yyrule14(yyarg1))
    case YYCHAR('{') => yygoto = 1; yynt = YYNTfeature_list(yyrule15(yyarg1))
    case DEF() => yygoto = 1; yynt = YYNTfeature_list(yyrule12(yyarg1))
    case OVERRIDE() => yygoto = 1; yynt = YYNTfeature_list(yyrule12(yyarg1))
    case VAR() => yygoto = 1; yynt = YYNTfeature_list(yyrule12(yyarg1))
    case _ =>  yygoto = 2; yynt = YYNTfeature_list(yyarg1);
  }
  } catch {
    case YYError(s) => yynt = YYNTerror(s);
  }
  while (yygoto == 0) {
    yynt match {
      case YYNTerror(s) =>
        yyerror(s)
        yypanic({ t:YYToken => t match {
          case YYCHAR(';') => true
          case _ => false
        }})
        yygoto = yystate14(yyarg1);
      case _:YYNTfeature_list => return 0;
    }
  }
  yygoto-1
}
```

Fig. 3. Generated Scala code for the LC state from Fig.2.

`error` pseudo-nonterminal). We simulate multiple-level returns for the recursive ascent parse functions by putting the return value in field `yynt` and returning the number of frames (`yygoto`) that must still be popped.

Figure 4 shows the recognition function (predictive parsing routine) for rule 12. This function is called when implementing an "announce" action (as seen in Fig. 3). It includes the semantic action: in this case translated from `{ $$ = $1 + $2; }`.

The final kind of generated function is the one that starts a recursive descent parse for a given nonterminal. Figure 5 shows the parsing function for the "feature" nonterminal. This routine is called from the code in Fig. 4. Such functions are not private so that they can be used by the code that interfaces with the generated parser.

The generated parser has a simple interface to the scanner: the parser is

```
/** Recursive descent parser after recognition point
 * feature_list: feature_list . feature ';'
 */
private def yyrule12(yyarg1 : Features) : Features = {
  var yyresult : Features = null;
  val yyarg2 : Feature = parse_feature();
  parse_YYCHAR(';');
  { yyresult = yyarg1 + yyarg2; }
  yyresult
}
```

Fig. 4. Recognition function for Rule 12.

```
def parse_feature() : Feature = {
  yystate17();
  yynt match {
    case YYNTfeature(yy) => yy
    case YYNTerror(s) => throw new YYError(s)
  }
}
```

Fig. 5. Sample parse routine for a nonterminal.

started by passing it an iterator that returns the tokens.

3 Related Work

The number of parser generators using LL or (LA)LR technologies is great. There are fewer tools generating recursive ascent parsers [5,10], and to our knowledge only Horspool has previously written a recursive ascent-descent parser generator.

The primary mechanism for text parsing included with the Scala standard library is that of parser combinators [6]. Parser combinators are an embedded DSL in Scala for expressing EBNF-like grammars. The executable code is generated directly by the Scala compiler, there is no need for an external tool (such as ScalaBison) to process the grammar description. At a very high level, parser combinators are a representation of LL(*) parsing without using tables. Instead, input is consumed by a `Parser`, which reduces to either `Success` or `Failure`, dependent upon whether or not the input was successfully parsed. In general, combinators use backtracking which impacts efficiency negatively. Grammars of arbitrary complexity may be represented by composing smaller parsers.

Generator	Compiled Size (K)	good.cl Time (ms.)	good.cl Space (MB)	large.cl Time (ms.)	large.cl Space (MB)
combinators	350	54	3.1	275	3.3
ScalaBison	200	17	0.2	36	2.1
Beaver	70	8	0.2	19	1.5

Table 1
Comparing ScalaBison with other generators.

4 Evaluation

One common concern with recursive ascent parsers is that the large number of states leads to a large code size. Indeed Veldema [10] decided against a purely direct-encoded parser for this very reason, opting instead for an approach that can be seen as table-driven. Recursive ascent-descent parsing is supposed to alleviate this problem and indeed we find that the number of states is noticeably fewer: about 40% fewer for grammars that make heavy use of LR features. For example, the published LALR(1) grammar of Java 1.1 (in which optional elements are expanded to split one production into two, resulting in a grammar with 350 productions) yields 621 states with bison but only 378 in ScalaBison. We also tested a grammar for a dialect of Cool [1] which made heavy use of precedence declarations (67 productions): 149 states for bison, 100 for ScalaBison. The reduction in states is to be welcomed but may not be great enough to make recursive ascent-descent attractive to people repelled by recursive ascent. The generated parser for Cool is still over 100K bytes of Scala, and for Java 1.1 over 600K bytes. By way of comparison, the bison generated parsers are 53K and 120K of C source respectively; bison does a good job compressing the tables and directly encoded parsers don't lend themselves as easily to compression.

To measure performance, we compare the ScalaBison Cool parser with one written using parser combinators. The comparison is not "fair" in that parser combinators were designed for clarity, not speed, and furthermore, the ScalaBison parser uses a hand-written (but simple and unoptimized) scanner whereas the combinator parser operates directly on the character stream. We also compared ScalaBison with Beaver (`beaver.sourceforge.net`), reported to generate the fastest LALR(1) JVM-based parsers.

Table 1 shows the results of testing Cool parsers implemented by all three generators against an Cool input file (`good.cl`) comprised of roughly 3,100 tokens exercising every production of the grammar. The file `large.cl` simply repeats this file ten times. The first column shows the compiled code size. The "Space" columns show the maximum memory usage ("high water mark")

during the runs. All tests were performed using a MacBook Pro, 2.4 Ghz Core 2 Duo with 4 GB of DDR3 memory using Apple's JDK 1.5.0_16 and Scala 2.7.3.final. Each test was run twelve times with the best and worst results dropped, and remaining ten times averaged. Garbage collection was triggered between each test.

Beaver generates noticeably faster code. Part of the difference is due to the fact that the numbers for ScalaBison include running the scanner which takes roughly half the reported time, whereas the (different) scanner uses up only a third of Beaver's much smaller time. However, even taking the scanner time out of the parse time still leaves Beaver's parser faster. One factor is that ScalaBison uses `match` clauses (see Figure 3) which the Scala compiler implements with a linear search, whereas Beaver uses dynamic dispatch.

5 Conclusion

ScalaBison is a practical parser generator for Scala built on recursive ascent-descent technology that accepts bison format input files. This enables the benefits of direct-encoding while reducing code size from a pure recursive-ascent solution. It uses bison's LALR(1) tables to build its own LC tables and thus is able to provide the same semantics of conflict resolution that bison does. The parsers generated by ScalaBison use more informative error messages than those generated by bison. The parsing speed and space usage are much better Scala's built-in parser combinators but are somewhat slower than the fastest JVM-based parser generators.

SDG

References

[1] Alexander Aiken. Cool: A portable project for teaching compiler construction. *ACM SIGPLAN Notices*, 31(7):19–26, July 1996.

[2] Alan J. Demers. Generalized left corner parsing. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, Los Angeles, California, USA, pages 170–182. ACM Press, New York, January 1977.

[3] R. Nigel Horspool. Recursive ascent-descent parsing. *Journal of Computer Languages*, 18(1), 1993.

[4] F. E. J. Kruseman Aretz. On a recursive ascent parser. *Information Processing Letters*, 29(4):201–206, 1988.

[5] René Leermakers. Non-deterministic recursive ascent parsing. In *Proceedings of the fifth conference on European chapter of the Association*

for Computational Linguistics, pages 63–68. Association for Computational Linguistics, Morristown, NJ, USA, 1991.

[6] Adriaan Moors, Frank Piessens, and Martin Odersky. Parser combinators in Scala. Technical Report CW491, Department of Computer Science, K.U.Leuven, February 2008.

[7] Thomas J. Pennello. Very fast LR parsing. *ACM SIGPLAN Notices*, 21(7):145–151, 1986.

[8] Paul Purdom and Cynthia A. Brown. Semantics routines and $LR(k)$ parsers. *Acta Informatica*, 14:299–315, 1980.

[9] G. H. Roberts. Recursive ascent: an LR analog to recursive descent. *ACM SIGPLAN Notices*, 23(8):23–29, 1988.

[10] Ronald Veldema. Jade, a recursive ascent LALR(1) parser generator. Technical report, Vrije Universiteit Amsterdam, Netherlands, 2001.

Library Concepts for Model Reuse

Markus Herrmannsdörfer [1] Benjamin Hummel [2]

Fakultät für Informatik
Technische Universität München
Garching bei München, Germany

Abstract

Reuse and the composition of libraries of partial system descriptions is a fundamental and well-understood practice in software engineering, as long as we talk about source code. For models and modeling languages, the concepts of reuse often are limited to *copy & paste*, especially when it comes to domain-specific modeling languages (DSLs). This paper attempts to give an overview of techniques for including support for reuse and library concepts both in the meta-model and the modeling tool, and presents a novel generative approach for this task. The technical consequences for each of the approaches presented are discussed and compared to each other.

Key words: reuse, model library, model-based development

1 Introduction

A major trend in software engineering is the increased use of semi-formal models during software development. This includes models for capturing requirements, describing the software's design and architecture, or capturing a more formal specification of a system. These models are then used for different analysis and generation tasks which are not possible using documents written in natural language – such as checking consistency of different views of a system, generating source code or test cases from a model, or formal verification. While UML [8] has been a key player for driving model-based development in software development, in many domains the usage of so-called domain-specific modeling languages is more appropriate. These range from simple languages for the user interface design of digital watches [6], to more complicated ones for defining business processes [9] or developing software for embedded systems, such as the well-known example of Matlab/Simulink [11].

[1] Email: herrmama@in.tum.de
[2] Email: hummelb@in.tum.de

With these models getting more and more important and growing in size, reuse of common sub models becomes an important issue. Reusing well-tested parts of a model reduces the risk of introducing bugs and reduces the overall development costs. In the long term identifying commonly used parts and putting them into a library for later use – possibly adding a mechanism for parametrization – has lots of advantages over the commonly found reuse by just *copy & paste*. First, it makes reuse more structured and systematic, as it makes explicit which parts are actually designed to be used elsewhere – compared to just looting existing models. Second, it allows to better organize analysis and testing activities, by creating libraries of well understood and tested elements. Finally, it hugely simplifies dealing with change, both due to shifted requirements or detected bugs. Without a library, all duplicates of the model part being changed have to be found beforehand, which despite existing approaches for automation [2] is a tedious and error-prone [4] task. Thus we argue that meta-models and editors for domain-specific modeling languages should support structured reuse and library concepts.

Problem Statement

Although the need for reuse at the model level is often confirmed, only little work is available on the patterns and concepts used to integrate reuse and library support into a modeling language. Especially the impact which different realizations might have on the underlying meta-model and the tools manipulating and analyzing the models is rarely discussed.

Contribution

This paper attempts to give an overview of the possible choices for implementing reuse and library support into a DSL's meta-model and tool chain. Therefore we summarize a "well-known" technique for this task as well as one suggested in the literature, and introduce a (to the best of our knowledge) novel generative approach for achieving it. Our focus is especially on the discussion of the impacts and consequences implied by choosing one of them. The ideas and insights are mostly rooted in the current development of AutoFOCUS 3[3], a reimplementation of the AutoFOCUS research prototype for modeling embedded systems [10], and a proprietary editor for the COLA language [7] which was developed within the context of an industry project from the automotive domain.

Outline

The next section introduces our running example, which is used to demonstrate the impact of adding support for reuse. Sec. 3 gives an overview of existing approaches, while Sec. 4 introduces our solution to this problem. In

[3] Actually, the concepts discussed here are part of the underlying framework called CCTS. Details are available at http://af3.in.tum.de/.

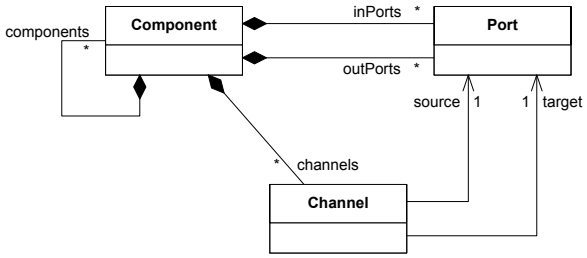


Fig. 1. Running Example Meta-model.

Sec. 5 we discuss the consequences implied by each of them, before we conclude in Sec. 6.

2 Running Example

We use a data-flow language for the component-oriented specification of systems as a running example throughout the paper. An excerpt of the language's meta-model is depicted in Fig. 1 as a UML class diagram. A Component defines a syntactic interface which consists of input and output Ports (compositions inPorts and outPorts). A component is either a basic component or composed of sub components, thus resulting in a component hierarchy. The sub components of a component are connected to each other via channels. A Channel of a component connects a source to a target port – with the restriction that only intra-level connections are allowed.

Reuse of components is a key technique to reduce the effort for the description of a system. Now we want to extend our modeling language in a way that it supports the reuse of components. In the following, components that can be reused are called component *types*. We should be able to make component types available through a so-called *library*. When it comes to reuse of a component, the reused component is called an *instance* of the component type. An instance of a component type can be used wherever a regular component can be used, *e.g.*, in the composition of another component. An instance has to be aware of its component type, so that it can adapt itself to changes of this component.

We deliberately kept the running example as simple as possible in order to be able to convey our ideas. However, the meta-models we developed in the context of our tools are more extensive, defining more than a hundred classes. This is due to the fact that the corresponding modeling languages also provide type systems, enable the specification of component properties, and allow us to model the system at different levels of abstraction, like requirements and technical realization [1]. As a consequence, reuse should not only be enabled for components, but also for different artifacts like types, requirements, or hardware components. Therefore, we need a generic reuse concept or pattern that is not only applicable to components, but also to different artifacts.

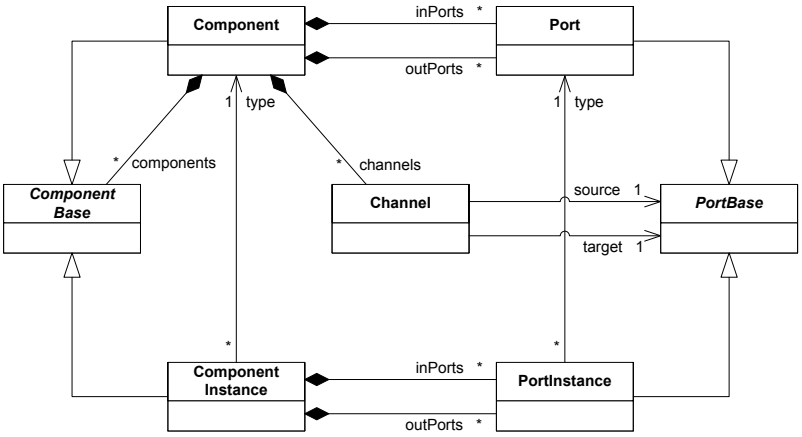


Fig. 2. Introduction of Instance Classes.

3 Existing Approaches

Literature on techniques for model reuse is rather scarce, so in this section, we concentrate on describing the two most prominent approaches to integrate reuse into a modeling language. The first approach solves the problem at the level of the meta-model by introducing new classes, whereas the second approach solves it at the level of the meta-meta-model by introducing a generic cloning mechanism. We detail the characteristics of either approach, apply them to our running example, and mention their implications for the meta-model and the modeling tools.

3.1 Introduction of Instance Classes

Reuse can be integrated by introducing classes that explicitly model the instances of types. This technique is presented in [3] where it is applied in a number of domain-specific meta-model patterns.

In the following, the introduced classes are called *instance classes*, whereas the classes of the types are called *type classes*. An association from an instance class to its corresponding type class allows an instance to be aware of its type. This association has to be many-to-one, as there is exactly one type for an instance, and there may be a number of instances for a type. As is depicted in Fig. 2, the class ComponentInstance is introduced to model instances of component types. The corresponding type class Component is accessible from the instance class ComponentInstance through the association type.

To allow an instance to be referenced from other elements, certain child elements of the type have to be instantiated, too. An instance then basically replicates a certain part of the structure of the type which we call *interface* in the following. Additional constraints have to be introduced to the meta-model

to ensure that the instance correctly replicates the interface of the type. In our example, the interface of a component also consists of input and output ports. As is depicted in Fig. 2, we thus have to introduce an instance class for ports, namely the class PortInstance, and compositions inPorts and outPorts between ComponentInstance and PortInstance. Furthermore, we have to add constraints to ensure that each component instance consists of an instance of each of the component's ports.

In order to be able to use instances in the same contexts as types, common super classes for instance and type classes are introduced. These common super classes have to be abstract, as their purpose is to enable instances and types to be used interchangeably. The associations that make up the context in which an element is used have to target these new super classes. In our example, we have to introduce new classes ComponentBase as a common super class of Component and ComponentInstance, and PortBase as a common super class of Port and PortInstance (see Fig. 2). The composition components is re-targeted to the class ComponentBase, so that component instances can also be used to compose components. Furthermore, the associations source and target are re-targeted to the class PortBase, so that port instances can also be connected by channels.

In a nutshell, this pattern requires that the meta-model is extended. For each type class of the type's interface, two new classes have to be introduced: the instance class and a common super class for both instance and type class. Additionally, the structure between the instance classes has to replicate the structure between the type classes. Furthermore, new constraints have to be added to guarantee conformance of an instance to its type. As the meta-model is modified, tools depending on the meta-model, *e.g.*, editors and interpreters, have to be adapted to cater for reuse. This also includes the capability to propagate changes of the interface of the type to its instances. Existing models do not have to be migrated, as the meta-model modification preserves existing model elements. However, future meta-model extensions which enlarge the interface of a type require to introduce new instance classes. When it comes to that, the migration of models is necessary, as additional elements have to be replicated for the existing instances.

We applied this technique to the COLA language [7] which was developed in the context of an industrial project. The instance classes were already integrated, before we started to develop tools for the language. Nevertheless, the tool's implementation grew more complex, as it had to take the additional instance classes into account. A later development step added an additional abstraction level, which required trace links to the existing components. As the composition for sub components was not part of the interface, the sub components of instances are not directly represented as model elements, and trace links thus could not target them. In order to be able to address such an instance anyway, we had to use the following workaround: we reference the instance through the path which leads from the root component to it along the

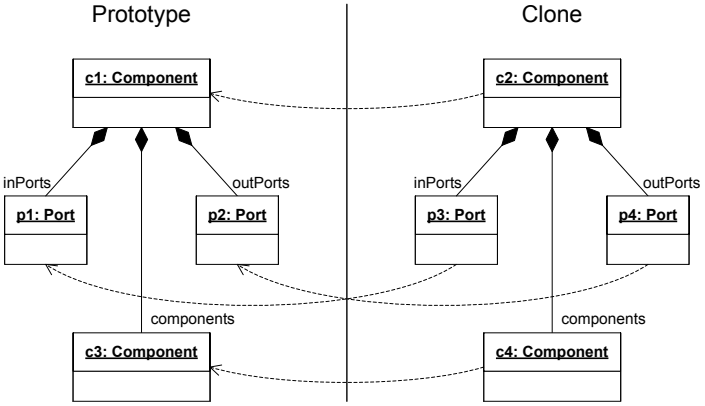


Fig. 3. Generic Cloning of a Component.

type/instance tree. This technique however decreased the overall simplicity of the meta-model, as these paths have to be kept consistent with the component structure.

3.2 Generic Cloning Mechanism

The previous technique requires only the interface of a type to be replicated. Another technique outlined in [5] is to fully replicate the internal structure of the type. This makes it possible to integrate reuse in a more generic manner at the level of the meta-meta-model. As a consequence, such capabilities have to be provided by the underlying meta-modeling framework. A reference implementation is provided by the authors through GME (Generic Modeling Environment [4]).

A type is basically instantiated by creating a copy of the element. As this technique is known from prototype-based programming languages, a type is called *prototype* and an instance is called *clone* in the following. Due to the meta-modeling framework, the clone is aware of the single prototype from which it is derived. As is depicted in Fig. 3 as a UML object diagram, the component `c1` is reused by producing a clone `c2`. The dashed line indicates that the clone `c2` is associated to its prototype `c1`.

A clone actually is a deep copy of a prototype, meaning that it also replicates the internal structure of the prototype. The clone's children are also aware of the prototype's children from which they are derived, and so forth. Fig. 3 shows that the clone not only copies the component, but also its ports and sub components.

A clone trivially has the same type as the prototype from which it is derived. As a consequence, it can easily be used in the same contexts as the

[4] http://www.isis.vanderbilt.edu/Projects/gme/

prototype. Therefore, a clone of a component is also of type Component and can thus be used to compose other components.

As a clone is aware of its corresponding prototype, changes to the prototype are automatically propagated to the clone. A clone can be modified independently of the prototype – changes to the prototype are then only propagated to the unmodified parts of the clone. In our example, the addition of a port to a component leads to the addition of a clone of that port to all of the component's clones. The clone of the component may be customized by adding a new port.

Due to its genericity, this technique does not require any extensions to the meta-model. However, the meta-modeling framework has to provide the capabilities for cloning model elements. To the best of our knowledge, such a feature currently only exists for GME. As cloning is implemented on the level of the meta-meta-model, model elements of all types can be cloned – without restriction. Due to methodological issues, cloning may only be allowed for certain types of model elements. Compared to the previous technique, types have to be completely replicated when instantiating them. In addition, information about which features of clones were overwritten or not has to be maintained within the model. This may excessively increase the amount of information which has to be stored in a model. Furthermore, generic cloning is subject to the following restriction: clones can only be generated from prototypes which do not use cloning in their internal structure. In case this restriction is not enforced, this may lead to clones which do have several prototypes. In Fig. 3, a clone could not be generated from component c1 if there were a cloning relation between ports p1 and p2, as port p4 then would have two prototypes – namely ports p1 and p3.

4 Generative Libraries

For the CCTS/AutoFOCUS 3 framework we started thinking about libraries, when the meta-model already consisted of nearly 100 classes and much of the core functionality, such as graphical editors, the expression evaluator, a simulator, and first parts of a code generator, were already implemented, resulting in more than 50.000 lines of source code. Thus the solution had to be minimally invasive, affecting the meta-model and the existing code as little as possible, as resources for going over the entire code base just were not available.

The overall idea was to design a library mechanism, which is orthogonal to the existing meta-model and modeling tools as far as possible. So for the existing code the change should be transparent by providing models that are compatible to those used before. This lead us to an approach similar to [5] where instances are actually full clones of the types. The main difference is that our focus is on a more structured reuse mechanism, where elements may not be arbitrarily cloned, but only dedicated *library elements* may be instanti-

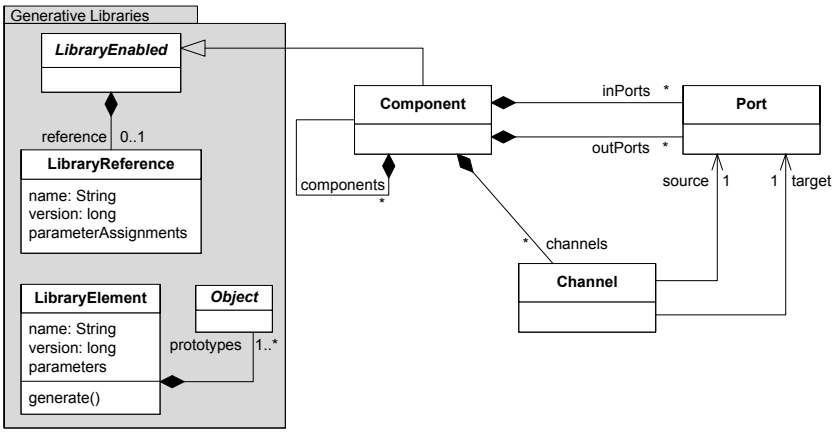


Fig. 4. Meta-model for Generative Libraries.

ated. In addition, our approach generalizes cloning in a way that allows us to parametrize instances that are actually generated by the type. In contrast to GME, where cloning is deeply integrated into the meta-modeling framework, our approach can be easily built on top of each existing meta-modeling framework. Our modeling tool for instance is implemented on top of EMF (Eclipse Modeling Framework [5]). It however replicates some of the features of GME in the implementation of the modeling tool. In the following, we outline the overall approach and provide more technical details and implications of our approach.

4.1 The Big Picture

In our setting we differentiate between the library, which contains types, and the importing model, which contains instances of some of these types. The association of an instance to its corresponding type is only performed by name (lazy linkage), which allows the implementation of different storage and look-up schemes. For example, we decided to keep libraries in separate files that can be individually imported by other models.

Using our approach, the meta-model grows by three classes [6], no matter for how many existing classes reuse should be enabled. These classes and their usage are shown in Fig. 4.

The abstract class LibraryEnabled is used to mark all meta-model classes which can be used as types. LibraryEnabled introduces a reference, which allows to distinguish "normal" model elements from instances. If the reference does not point to a type in the library, it is a plain element, otherwise the associated

[5] http://www.eclipse.org/modeling/emf/

[6] Depending on how the library is organized internally and which mechanism is used for parametrization, there may be more classes for library folders and parameters. The growth in meta-model elements however is constant.

LibraryReference stores the name of the referenced type, the version used for the last update, and the parameter assignments used for generating the current instance. In our example, only a Component can be reused via libraries, but for larger meta-models multiple elements can easily be marked to serve as types.

The class LibraryElement is the container for types in a library and defines a name, a version number which has to be incremented with each change of this element, and a set of supported parameters. Most important are the prototypes, which are stored in the LibraryElement. The prototypes composition may include any modeling element (indicated in Fig. 4 by the use of Object which by definition is the transitive super class of all meta-model classes), which can then be used for generating instances of this element. The LibraryElement acts as a factory for a certain kind of class (which is determined by the first prototype in the list) by means of its generate() method which generates an instance based on the parameter assignments. To explain how the meta-model classes are actually used, we explain the three use cases of creating, instantiating, and modifying library elements.

Creation of a library element

The creation of a new type consists of the creation of a LibraryElement, which then is composed of the required prototypes. What information the LibraryElement uses to create an instance from given parameters is up to the implementation. The most basic version just returns a copy of its first prototype, which corresponds to the cloning approach presented in Sec. 3.2. In our tools we also experimented with the inclusion of Groovy [7] scripts, which then generate a new instance using these prototypes based on the given parameters. In our example, this might be the number of ports and the kind of sub components of a component. There are no further limitations concerning the generation, except that a library element always has to return an instance of the same class as generation result (in our case a Component).

Instantiation of a library element

To instantiate a library element for reuse in an existing model, the generate() method of the corresponding element is invoked and the result is inserted into the model. Hence, the generated instance is completely stored in the model, which ensures that the model has the same structure as if built without library support. This is crucial to avoid complicated changes in existing code as mentioned before. The generated element is completed by attaching a LibraryReference which stores the name and version of the library element used for creation. A side effect of the inclusion of plain copies is that the model has all required information even in the absence of the libraries it depends on.

[7] http://groovy.codehaus.org/

Modification of a library element

The most important reason for building library mechanisms is to support the modification of a library element, which should then be reflected by all of its instances. As we model reuse only by marking instances with their origin and library references are only loosely coupled by their name, we have to deal with certain aspects of reuse in the implementation of our modeling tool. The first part is that the version number of a library element has to be increased with every change to it. As these changes are performed using the modeling tool, this should not be problematic. Using the version number, we can easily identify LibraryReferences, which are outdated (*i.e.*, point to an earlier version of a library element) and thus have to be replaced by an updated version. In our implementation this check is triggered manually by the user to allow full control of the modification of an existing model, as changes in a library are a potential threat to the correctness of a model. However, depending on the tooling infrastructure, such a task could also be performed automatically on certain events. As the prototypes of a library element may again be library instances, the check and update procedure has to be performed in a recursive fashion.

The more interesting part is the update of all instances of a library element. Actually, we have to replace a part of the model by a new one, which was generated by the generate() method. A simple exchange of the model parts is not sufficient here, as an instance might include elements which are referenced from other parts of the model (*e.g.*, a channel to one of its ports) or layout and naming information, which we do not want to loose. Our solution uses a modified merge algorithm, which replaces the old model part with the new one, but retains all references to external (relative to the part being exchanged) elements. Using modern reflective meta-modeling frameworks such as EMF, this can be achieved in a generic fashion using only a small amount of code (our implementation has about 300 code lines including comments). However, this piece of code has to be slightly adjusted for the meta-model used, as it has to treat some data differently. For example in AutoFOCUS 3, the name of the top-most element in an instance may be changed by the user and should not be overwritten by a change in the library.

4.2 Technical Considerations and Implications

Using the scheme described so far, most parts of the modeling tool's code can be left untouched. All that is needed is new code for creating and managing the libraries and for handling the update of instances after modifications to their corresponding library elements. There are some additional minor modifications not described here. For example one usually wants to view elements in the hierarchy of a library instance, but it should not (or only in some limited way) be allowed to change them, as those changes would be lost after updating the instance from the library. So the editors have to support (par-

tially) read-only parts of the model. All in all our implementation, which is based on EMF, has about 3.000 lines including the code for managing libraries (folders, etc.) which was not described here. Especially tasks such as updating the version number of a library element upon modification are trivially implemented using recursive listeners which react to all changes to an element *or* its children. The meta-model specific parts of the implementation could be limited to the merge algorithm, though it was kept general enough to make an update only required for severe meta-model changes, such as changing the way layout data is stored. Reuse of the code in other applications is still difficult, as most of it is specific for our modeling tool (*e.g.*, how to switch an editor to read-only mode, etc.).

All in all the implementation and modification effort for our solution was quite manageable, especially as the full (rolled out) model is available and the identification and referencing of individual elements is easy. However, our approach also has some drawbacks. The most obvious is the increased size of the models, as redundant information is stored. While in theory the growth in the number of model elements can be exponential, in our experience the elements being reused are often relatively generic and small, so their number is not a major problem. The larger elements being reused usually solve specialized tasks and thus are used only a couple of times (*e.g.*, a component for monitoring the wheel pressure of a car). Our estimates rather indicate a growth by a constant factor, which depends on the actual model but is in the magnitude of 2 to 10. With the disk space and memory size available today this often is not a big deal, but might be an issue for specific applications or extremely large models.

Another problem with our approach is the slightly concealed reuse structure. While similar or identical parts of the model can easily be determined by looking at the LibraryReferences, their exploitation, *e.g.*, generating the code for these parts only once, can be more complicated than with explicit instance classes. This is especially true in the presence of parametrization.

5 Discussion

To ease the selection of a reuse mechanism and clarify the differences of the presented approaches, in this section we discuss their strengths and weaknesses along several criteria. A summary of this discussion is presented in Tab. 1 which is explained in more detail in the remainder of this section. We want to stress that there is no "best" approach. Depending on the requirements and context each of the presented approaches might be the most suitable. Also the importance of the criteria will be different for every application, so simply "summing up the pluses" is not a valid evaluation. Rather it should be assessed whether there is a hard knock-out criterion (*e.g.*, if the meta-modeling framework is fixed and is not GME, generic cloning might not be a good choice), and then the remaining criteria should be prioritized and

Criteria	Instance Classes (Sec. 3.1)	Generic Cloning (Sec. 3.2)	Generative Libraries (Sec. 4)
Independence of meta-modeling framework	+	–	+
Growth in meta-model size	–	+	+
Size of models	+	–	–
Deep references	–	+	+
Implementation effort	–	+	0
Reuse strategy	planned	ad hoc	planned
Arbitrary parametrization mechanisms	–	–	+
Automatic update of instances	0/-	+	0/+

Table 1
Comparison of Reuse Approaches

evaluated.

Independence of meta-modeling framework

Our first criterion is to what extent the approach is independent of the meta-modeling framework used. Clearly the approach using instance classes can be applied in the context of any meta-model, as it only uses standard meta-modeling constructs. This also applies to our generative approach, as we do not rely on special features of the meta-modeling framework. Only the implementation might be slightly more complex, if the meta-modeling framework does not support reflection. In contrast, generic cloning is currently only implemented in GME and the extension of a different meta-modeling framework to support generic cloning is a non-trivial task.

Growth in meta-model size

The size and complexity of the meta-model affects the time and effort required for developers to understand and use it. Consequently larger meta-models are more likely to be used in a fashion which was not intended. Additionally, if the meta-model is to be supported by constraints, the number and complexity of these constraints also usually increases with the size of the meta-model. As seen in the previous examples, a number of classes have to be introduced for each kind of type when using instance classes. In contrast, the generative approach requires only a constant number of classes to be introduced, no matter how many classes are marked as types in the meta-model. In generic cloning the same effect is achieved by extensions to the meta-meta-model.

Size of models

As discussed earlier, the size of a model might not be the most critical factor these days, as disk and memory is available at little cost. Still there are applications where the models can reach sizes, which demand a space-efficient representation of the model. These applications might profit from using instance classes, where redundancy in the model is reduced to a minimum. Both of the other approaches store and manipulate a fully expanded model, where all instances are actually full copies of their types. Generic cloning even requires additional storage for tracking changes to the features of all clones.

Deep references

For many applications we have to reference model elements, which are part of an instance. Examples include requirements tracing or deployment models. Creating those deep references is quite easy for generic cloning and generative libraries, as all parts of an instance are realized in the model and thus can be referenced. This is different for instance classes, where these parts are only represented once in the corresponding types. Solutions then either use more complicated constructs for referencing these elements (such as paths along the type/instance tree), or the instances have to be partially realized in the model. In our example, the ports have been replicated for the instance, to allow the channels to connect to them. It is however more complicated to reference the sub components of a component instance.

Implementation effort

Another important factor is the effort required when implementing editors and tools working with these models. The least effort is required with the generic cloning approach when using the GME framework, as for the tools it is completely transparent whether they are working on a prototype or a clone. All cloning and reuse issues are dealt with by GME. This is nearly the same for generative libraries, but the management of types and their instances has to be handled by our tools. As described earlier, the implementation effort for this is still manageable and the editors still do not need adjustment. When using instance classes, the tools dealing with the model, such as editors and simulators, have to cope with the additional instance classes.

Reuse strategy

To some extent, the reuse approach affects the reuse strategy available. We differentiate planned reuse, where elements are explicitly marked or modeled as a type which can be reused by instantiation, and ad hoc reuse, where each element in the model may be used as a prototype. Which one is preferred, depends on the domain used. In embedded systems development, for instance, planned reuse is often favored, as components released in a library might have to be tested and documented more rigorously. In other situations ad hoc

reuse might be more suitable, as the modeler does not have to interrupt the modeling flow by deciding which parts to put into a library. Instance classes and generative libraries are more designed toward planned reuse, while generic cloning better supports ad hoc reuse. However, the differentiation between these is somewhat blurry, as of course ad hoc reuse can be restricted by the modeling tools to forge it into a more planned fashion, and automatic creation of library types can be implemented to make the application of planned reuse more "agile".

Arbitrary parametrization mechanisms

Both with instance classes and generic cloning only rather simple parametrization mechanisms are supported. Typically an instance is just a duplicate of the type with possibly some attributes depending on parameters. With our generative approach, where the instance is generated by a method of the library element, any feature of an instance, including the model structure of the instance element, may be affected by parameter choice. This also simplifies the inclusion of different parametrization concepts and thus eases experimentation here.

Automatic update of instances

The crucial aspect when using reuse mechanisms is the update of the instances whenever the type changes. While on the first thought this seems to come at no cost for instance classes, it is actually the most expensive there. In our example, the sub components of an instance are updated automatically, as they are only present in the type, but the port instances always have to be updated, when the ports of the component type change. The situation is similar with generative libraries, but as we always duplicate the entire instance, the update/merge algorithm can be kept more generic, which dramatically simplifies its implementation. With generic cloning the update is deeply integrated into the meta-modeling framework.

6 Conclusion

This paper provided an overview of different approaches to integrate reuse into models and modeling languages. The focus was especially on the consequences on the meta-model and the implementation of modeling tools. For cases where existing modeling languages are to be extended by library mechanisms, a generative approach was presented, which requires only little modifications to the meta-model and existing code.

To aid in the selection of one of these approaches, we discussed their strengths and weaknesses along several dimensions. None of the reuse approaches clearly dominates the others, but given a specific context and requirements often simplifies the selection. As our discussion reveals, the "classical" solution using type and instance classes often unnecessarily complicates the

implementation of tools. In contrast to textual modeling (like source code), controlled cloning seems to be beneficial in many cases.

References

[1] M. Broy, M. Feilkas, J. Grünbauer, A. Gruler, A. Harhurin, J. Hartmann, B. Penzenstadler, B. Schätz, and D. Wild. Umfassendes Architekturmodell für das Engineering eingebetteter Software-intensiver Systeme. Technical Report TUM-I0816, Technische Universität München, 2008.

[2] F. Deissenboeck, B. Hummel, E. Juergens, B. Schaetz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *Proc. 30th Int. Conf. on Software Engineering (ICSE'08)*, pages 603–612. ACM, 2008.

[3] M. Fowler. *Analysis patterns: reusable objects models.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[4] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proc. of the 31st International Conference on Software Engineering (ICSE'09)*, 2009. To appear.

[5] G. Karsai, M. Maroti, A. Ledeczi, J. Gray, and J. Sztipanovits. Composition and cloning in modeling and meta-modeling. *IEEE Transactions on Control Systems Technology*, 12(2):263–278, March 2004.

[6] S. Kelly and R. Pohjonen. Domain-specific modelling for cross-platform product families. In *Proc. Conceptual Modeling - ER 2002, 21st International Conference on Conceptual Modeling (Workshops)*, pages 182–194. Springer, 2002.

[7] S. Kugele, M. Tautschnig, A. Bauer, C. Schallhart, S. Merenda, W. Haberl, C. Kühnel, F. Müller, Z. Wang, D. Wild, S. Rittmann, and M. Wechs. COLA – the component language. Technical Report TUM-I0714, Technische Universität München, 2007.

[8] Object Management Group. Unified Modeling Language, Superstructure, v2.1.2, 2007.

[9] Object Management Group. Business Process Modeling Notation, v1.1, 2008.

[10] Bernhard Schätz and Franz Huber. Integrating formal description techniques. In *Proc. of FM'99, World Congress on Formal Methods*, pages 1206–1225. Springer, 1999.

[11] The MathWorks, Inc. *Simulink 7 User's Guide*, 2008.

Integrating Textual and Graphical Modelling Languages

Luc Engelen Mark van den Brand

Department of Mathematics and Computer Science
Eindhoven University of Technology (TU/e),
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands

Abstract

Graphical diagrams are the main modelling constructs offered by the popular modelling language UML. Because textual representations of models also have their benefits, we investigated the integration of textual and graphical modelling languages, by comparing two approaches. One approach uses grammarware and the other uses modelware. As a case study, we implemented two versions of a textual alternative for Activity Diagrams, which is an example of a *surface language*. This paper describes our surface language, the two approaches, and the two implementations that follow these approaches.

1 Introduction

Many Eclipse-based modelling formalisms focus on notations that are either mainly textual of mainly graphical. Although tools exist that transform models written in a textual language to representations of those models that can be manipulated and depicted using graphical notations, the construction and manipulation of models written using a combination of both languages is not well facilitated.

The popular modelling language UML offers graphical diagrams for the construction of models. Research has shown, however, that graphical languages are not inherently superior to textual languages [16] and that both types of languages have their benefits. Therefore, we investigate the integration of textual and graphical languages, to be able to exploit the benefits of both types of languages.

One of the problems that arise when using two or more languages to construct one model is that parts of the model written in one language can refer to elements contained in parts written in another language. Transforming a model written in multiple languages to a model written in one language involves introducing correct references between various parts of the model.

Existing tools are aimed at converting textual models conforming to grammars into models conforming to metamodels and vice versa [7,3]. These tools can not transform models that consist of parts that conform to grammars as well as parts that conform to metamodels.

We use a textual alternative for activity diagrams, a textual surface language, as a case study and have implemented two versions of this language. One alternative uses tools and techniques related to grammars, and the other uses tools and techniques related to models and metamodels. The approach related to grammars transforms UML models containing fragments of behaviour modelled using our surface language to plain UML models by rewriting the XMI representation of the model provided as input. We used the ASF+SDF Meta-Environment [20] to implement this approach. The approach related to models and metamodels extracts the fragments of surface language, converts them to metamodel based equivalents, transforms these equivalents to Activities, and uses these to replace the fragments in the original model. We used the openArchitectureWare platform [23] to implement this approach.

The remainder of this paper is organized as follows: Section 2 introduces a number of relevant concepts. A specification of the surface language we implemented, and a description of its embedding in the UML and the transformation from surface language to Activities is given in Section 3. The approach based on grammars is described in Section 4, and the approach based on models and metamodels is described in Section 5. A number of other applications involving the integration of textual and graphical languages, and the transformation of models constructed using multiple languages are discussed in Section 6. Section 7 discusses how our work relates to earlier work. We draw conclusions and discuss future work in Section 8.

2 Preliminaries

The surface language we present is a textual alternative for the activity diagrams of the UML. In this section, we give a brief description of Activities and explain what a surface language is.

We use the naming convention used by the OMG in the definition of the UML [12] when discussing concepts of the UML. This means that we use medial capitals for the names of these concepts.

2.1 UML Activities

Activities are one of the concepts offered by the UML to specify behaviour. Some aspects of an Activity can be visualized in an activity diagram. The leftmost part of Figure 1 shows an example of such a diagram.

An Activity is a directed graph, whose nodes and edges are called ActivityNodes and ActivityEdges. There are a number of different ActivityNodes, such as ControlNodes (depicted by diamonds) and Actions (depicted by

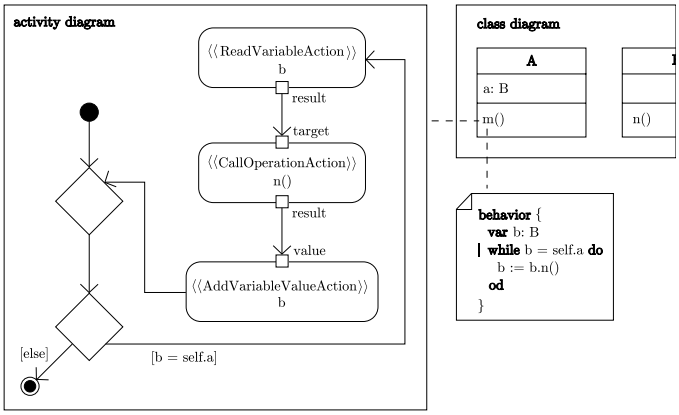


Figure 1. Two representations of the same behaviour

rounded rectangles), and two types of ActivityEdges, namely ControlFlows and ObjectFlows.

The informal description of the semantics of Activities states that the order in which Actions are executed is based on the flow of tokens. There are two kinds of tokens: control tokens and object tokens. ControlFlows, which are depicted by arrows connecting ActivityNodes, show how control tokens flow from one ActivityNode to the other. ObjectFlows, which are depicted by arrows connecting OutputPins and InputPins, show how object tokens flow from one Action producing an object to another Action that uses this object.

The ObjectFlows in Figure 1 are depicted by the arrows connecting the small rectangles on the borders of the Actions. These small rectangles are the InputPins and OutputPins of those Actions.

2.2 Surface Languages

Every model conforms to a metamodel, which defines the elements that play a role in the model. If a model conforms to a certain metamodel, each element of the model is an instance of an element in that metamodel. The UML defines a number of diagrams, which can be used to depict certain parts of a model. There are diagrams that depict the structure of a model, diagrams that depict the behaviour of parts of the model, etc. These diagrams offer a graphical representation for instances of elements in the metamodel.

A surface language offers an alternative notation for these diagrams. In our case, instead of a graphical representation, a textual representation is given for instances of elements of the metamodel. Other names for surface languages are *surface action languages* or *action languages*.

3 Specification of the Surface Language

To define our surface language, we must specify its syntax, semantics and embedding in the UML. The syntax of the surface language and its embedding in the UML are described below. The semantics of the language is defined implicitly by describing the transformation from behaviour specified in our surface language to Activities.

3.1 Syntax

The syntax of behaviour modelled in our surface language is as follows:

$$
\begin{aligned}
SLB &::= \text{“behavior”}\ \text{“\{”}\ [MVD]\ MS\ \text{“\}”}\\
MVD &::= \text{“var”}\ VD\ \{\text{“;”}\ VD\}\ \text{“|”}\\
VD &::= VN\ \text{“:”}\ TN\\
MS &::= S\ \{\text{“;”}\ S\}\,,
\end{aligned}
$$

where VN denotes a set of variable names and TN denotes a set of type names. A description of behaviour consists of a sequence of variable declarations and a sequence of statements. A variable declaration consists of a variable name and a type name.

The syntax of statements is as follows:

$$
\begin{aligned}
S ::= &\ \text{“if”}\ SWR\ \text{“then”}\ MS\ \text{“fi”}\\
\mid &\ \text{“if”}\ SWR\ \text{“then”}\ MS\ \text{“else”}\ MS\ \text{“fi”}\\
\mid &\ \text{“while”}\ SWR\ \text{“do”}\ MS\ \text{“od”}\\
\mid &\ \text{“return”}\ SWR\\
\mid &\ SN\ \text{“(”}\ [MSWR]\ \text{“)”}\ \text{“to”}\ SWR\\
\mid &\ SWR\ \text{“.”}\ ON\ \text{“(”}\ [MSWR]\ \text{“)”}\\
\mid &\ SWR\ \text{“.”}\ SFN\ [\text{“[”}\ \mathbb{N}\ \text{“]”}]\ \text{“:=”}\ SWR\\
\mid &\ VN\ [\text{“[”}\ \mathbb{N}\ \text{“]”}]\ \text{“:=”}\ SWR,
\end{aligned}
$$

```
<packagedElement xmi:type="uml:Class" name="C">
  <ownedBehavior xmi:type="uml:OpaqueBehavior" name="b">
    <language>SL</language>
    <body>return self</body>
  </ownedBehavior>
</packagedElement>
```

Figure 2. Embedding in the UML of behaviour modelled using a language called 'SL'

where *SN* denotes a set of signal names, *ON* denotes a set of operation names, *SFN* denotes a set of structural feature names and $\mathbb{N}$ denotes the set of natural numbers.

The syntax of statements with results is as follows:

$$
\begin{aligned}
\mathit{MSWR} ::= {} & \mathit{SWR}\ \{\text{“,”}\ \mathit{SWR}\} \\
\mathit{SWR} ::= {} & \text{“create” “(” } \mathit{CN} \text{ “)”} \\
| {} & \text{“self”} \\
| {} & \mathit{VN} \\
| {} & \mathit{SWR} \text{ “.” } \mathit{SFN} \\
| {} & \mathit{SWR} \text{ “.” } \mathit{ON} \text{ “(” } [\mathit{MSWR}] \text{ “)”},
\end{aligned}
$$

where *CN* denotes a set of class names, *VN* denotes a set of variable names, *ON* denotes a set of operation names and *SFN* denotes a set of structural feature names. Operation calls are listed both as statements and as statements with results, because both types of operation calls exist.

The note below the class diagram in Figure 1 shows an example of behaviour modelled using our surface language. The behaviour is equivalent to the behaviour represented by the activity diagram on the left of the figure.

3.2 Embedding in the UML

We use a concept of the UML called OpaqueBehavior to embed our surface language in the UML. Figure 2 shows a fragment of an XMI representation of a UML model that contains an instance of OpaqueBehavior.

OpaqueBehavior uses a list of text fragments and a list of language names to specify behaviour. The first list specifies the behaviour in one or more textual languages and the second list specifies which languages are used in the first list. OpaqueBehavior can be used to specify behaviour using, for instance, fragments of Java code or natural language. In our case, the first list contains a specification of behaviour using our surface language and the second list indicates that we use this surface language.

We transform a UML model containing behaviour modelled using our surface language to a UML model without such behaviour, by replacing all these occurrences of surface language embedded in OpaqueBehavior by equivalent Activities.

3.3 Transformation

As described in Section 3.1, behaviour specified using our surface language consists of two parts: a sequence of variable declarations and a sequence of statements. The process of transforming behaviour modelled using a surface language to an Activity can be divided into two steps:

(i) The variable declarations are translated to UML Variables.

(ii) The sequence of statements is translated to an equivalent group of ActivityNodes and ActivityEdges.

Translating variable declarations to UML Variables is a trivial step, which we will not discuss. An informal description of the second step is given below.

3.3.1 Transformation Function

We describe the transformation of sequences of statements to equivalent fragments of UML Activities by means of the transformation function T_B. The function T_B uses the auxiliary transformation functions T_{MS}, T_S and T_{SWR}. Figure 3 gives a schematic representation of the transformations performed by these functions.

The clouds and the dashed arrows in the figure indicate how the fragments are joined together to create an Activity. Each cloud in a fragment is replaced by another fragment of an Activity. An incoming dashed ActivityEdge shows how a fragment is connected to an outgoing ActivityEdge of the containing fragment; an outgoing dashed ActivityEdge shows how a fragment is connected to an incoming ActivityEdge of the containing fragment.

The function T_B creates a group of ActivityNodes and ActivityEdges that is equivalent to the sequence of statements provided as input, and connects this group with an InitialNode and an ActivityFinalNode using two ControlFlows.

The function T_{MS} creates an equivalent group of ActivityNodes and ActivityEdges for each of the statements in the sequence provided as input, and connects these groups using ControlFlows.

The function T_S creates a group of ActivityNodes and ActivityEdges that is equivalent to the statement provided as input. Statements with or without results that are part of the statement provided as input are also translated to equivalent groups of ActivityNodes and ActivityEdges. These groups are connected to the first group using ControlFlows, for statements, or ObjectFlows, for statements with results.

The function T_{SWR} creates a group of ActivityNodes and ActivityEdges that is equivalent to the statement with result provided as input. Statements with results that are part of this statement are also translated to equivalent groups of ActivityNodes and ActivityEdges. These groups are connected to the first group using ObjectFlows.

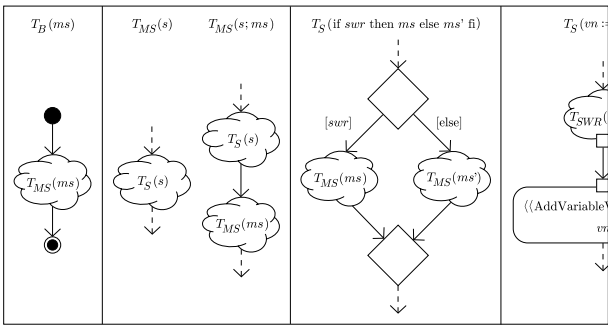


Figure 3. The transformations performed by the functions T_B, T_{MS}, T_S and T_{SWR}

4 Grammarware

In this section, we describe the implementation of our surface language that uses a tool for text-to-text transformations. Tools for text-to-text transformations are often referred to as *grammarware*. We start by describing our approach in Section 4.1. Section 4.2 describes the tools we used for the implementation and some important aspects of the implementation.

4.1 Approach

The leftmost part of Figure 4 gives a schematic overview of the transformation process when performed using a text-to-text (T2T) transformation.

The goal of this process is to transform a UML model containing behaviour specified using a surface language to a plain UML model. In the approach using grammarware, we transform models containing fragments of surface language to plain UML models by transforming the XMI [11] representations of those models. This transformation from one textual representation to the other consists of two steps:

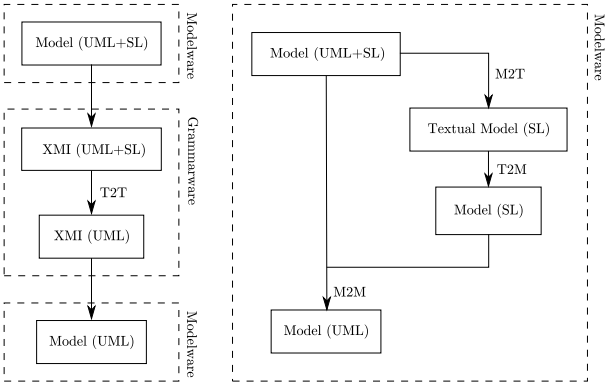


Figure 4. Two ways of incorporating textual languages in the UML

(i) A mapping from names occurring in the model to XMI identifiers is made, by traversing the parse tree of the XMI representation of the original model, and storing each name and the corresponding identifier in a table.

(ii) The transformation described in Section 3.3 is performed, by translating fragments of surface language to XMI representations of equivalent Activities.

The first step of the transformation makes it possible to retrieve the identifier of an element in the second step. Each element in the XMI representation of a UML model has a unique identifier. Actions that refer to other elements, such as AddVariableValueActions and CreateObjectActions, refer to these other elements using their identifiers. An AddVariableValueAction refers to a Variable using the identifier of that Variable; a CreateObjectAction refers to a Classifier using the identifier of that Classifier.

Grammarware has been a subject of research for quite some time. An

advantage of using grammarware to perform this transformation is the ease of use provided by the maturity of the tools and their documentation.

A disadvantage of transforming models using a text-to-text transformation is that the models have to be exported to a textual format. Having to deal with the textual representation of a model lowers the level of abstraction of the transformation. In our case, for instance, the transformation deals with concepts of the XMI language, at a low level of abstraction, instead of concepts of the UML, at a higher level of abstraction.

4.2 Implementation

We implemented the transformation described in Section 3.3 following the approach described in Section 4.1 in the language ASF+SDF [21], using an IDE for that language, called the Meta-Environment. We give a short description of this language and discuss some of the details of our implementation below.

4.2.1 ASF+SDF and the Meta-Environment

The language ASF+SDF is a combination of the two formalisms ASF and SDF. SDF stands for *Syntax Definition Formalism*. It is a formalism for the definition of the syntax of context-free languages. ASF stands for *Algebraic Specification Formalism*. It is a formalism for the definition of conditional rewrite rules. Given a syntax definition in SDF of the source and target language, ASF can be used to define a transformation from the source language to the target language.

Context-free languages are closed under union and, as a result of this, the SDF definitions of two languages can be combined to form the definition of a new context-free language, without altering the existing definitions.

Using ASF in combination with SDF to implement transformations guarantees syntax safety. A transformation is syntax safe if it only accepts input that adheres to the syntax definition of the input language and it always produces output that adheres to the syntax definition of the output language.

Both SDF and ASF specifications can be exported by the Meta-Environment. The exported SDF specification can then be used by command line tools to produce parse trees and transform parse trees to text. The exported ASF specification can be compiled to a fast command line tool suited for the transformation of such parse trees.

4.2.2 Implementation Details

An advantage of using SDF to define the syntax of our surface language is that it enabled us to combine this definition with an existing syntax definition of the XMI format, without any alterations to the definitions. This is due to the previously mentioned fact that context-free languages are closed under union.

Because transformations implemented in ASF are syntax safe, the transformation from UML models containing fragments of surface language to plain

```
$VariableName := $ReadVariableAction,
variableExists($Context, $VariableName) == true,
$Val := getVariableId($Context, $VariableName),
<$Id1, $Context1> := newId($Context),
<$Id2, $Context2> := newId($Context1),
$ObjectContent* :=
<node xmi:type="uml:ReadVariableAction" xmi:id=$Id1
variable=$Val>
  <result xmi:id=$Id2 />
</node>
====>
statementWithResult2Action($ReadVariableAction, $Context) =
  <$ObjectContent*, $Id2, $Id1, $Context2>
```

Figure 5. An ASF equation that creates a ReadVariableAction

UML models only produces results that adhere to the definition of XMI.

A disadvantage of the current implementation is that it can only parse one variant of XMI. Most tools that import or export files in the XMI format use their own interpretation of the format. These vendor specific interpretations are often incompatible with other interpretations. Because of this, our implementation is limited to XMI files produced by the UML2 plug-in of Eclipse, since it can only read and produce models that adhere to the interpretation of XMI of that plug-in.

A solution for this problem would be to introduce an intermediate language that serves as the starting point of a number of transformations to variants of XMI. We could then transform a model containing fragments of surface language to this intermediate language and subsequently from this intermediate language to a number of variants of XMI.

The limited portability is another disadvantage of using the Meta-Environment for the implementation of our approach, since it is currently only available for the Unix family of operating systems.

Figure 5 shows a part of the implementation in ASF of the transformation from behaviour modelled using our surface language to Activities. All variable names in this figure start with a dollar sign. The figure shows that a table mapping names to identifiers, denoted by the variable `$Context`, is used both to retrieve the identifier that corresponds to a given name as well as create fresh identifiers. The figure shows that every ReadVariableAction encountered in a fragment of surface language is replaced by the XMI in lines 7 to 9. Figure 6 shows the part of the SDF definition that defines the syntax of the surface language statement representing a ReadVariableAction and declares the corresponding variables. Line 4 of this definition defines that a ReadVariableAction is denoted by the name of a variable, as is specified in Section 3.1.

```
sorts
  ReadVariableAction
context-free syntax
  VariableName -> ReadVariableAction
variables
  "$ReadVariableAction"[0-9]* ->
ReadVariableAction
```

Figure 6. An SDF definition that defines the statement representing a ReadVariableAction

5 Modelware

This section describes the implementation of our surface language using tools for model-to-text, text-to-model and model-to-model transformations. Tools that can perform transformations related to models are often referred to as *modelware*. Section 5.1 describes our approach. Section 5.2 describes the tools we used for the implementation and some important aspects of the implementation.

5.1 Approach

The rightmost part of Figure 4 gives a schematic overview of the approach using model-to-text (M2T), text-to-model (T2M) and model-to-model (M2M) transformations within a UML modelling tool.

The process of using modelware to transform a UML model containing fragments of surface language to a plain UML model can be divided into the following steps:

(i) The fragments of surface language are extracted from the original model.

(ii) The extracted fragments are parsed and converted to a format usable by tools for model-to-model transformations.

(iii) The extracted and converted fragments of surface language are translated to equivalent Activities, as described in Section 3.3.

(iv) The fragments of surface language in the original model are replaced by the Activities created in the previous step.

An advantage of this approach is that all transformations can be performed from within one and the same modelling environment. In contrast to the approach described in Section 4.1, no models have to be imported or exported during the transformation process.

5.2 Implementation

We used three tools for model transformation from the openArchitectureWare platform to implement the transformation described in Section 3.3 following the approach described in Section 5.1. We describe these tools and the imple-

```
behavior b C {
  return self
}
```

Figure 7. An extracted fragment of surface language

mentation below.

5.2.1 Xpand, Xtend, Xtext and openArchitectureWare

The openArchitectureWare platform offers a number of tools related to model transformation: Xpand is used for model-to-text transformations, Xtext [3] is used for text-to-model transformations and Xtend is used for model-to-model transformations. Xpand and Xtend are based on the same type system and expression language. The type system offers simple types, such as *string*, *Boolean* and *integer*, collection types, such as *list* and *set*, and the possibility to import metamodels. The expression language offers a number of basic constructs that can be used to create expressions, such as *literals*, *operators*, *quantifiers* and *switch expressions*.

Xpand is a template-based language that generates text files given a model. An Xpand template takes a metaclass and a list of parameters as input and produces output by executing a list of statements. There are a number of different type of statements, including one that saves the output generated by its statements to a file and one that triggers the execution of another template.

Xtext is a tool that parses text and converts it to an equivalent model, given a grammar describing the syntax of the input. Xtext uses ANTLR [15] to generate a parser that parses the textual representations of models. An Xtext specification consists of rules that define both a metamodel and a mapping from concrete syntax to this metamodel. Given a specification of a textual representation, Xtext also generates an editor that provides features such as syntax highlighting and code completion.

Xtend is a functional language for model transformation. It adds *extensions* to the basic expression language, which take a number of parameters as input and return the result of an expression.

5.2.2 Implementation Details

We use Xpand to extract fragments of surface language from models by traversing these models. For each instance of OpaqueBehavior in a model, the string describing its behaviour is stored in a text file, including the name of the OpaqueBehavior and the name of the Class it is contained in. Figure 7 shows the fragment of surface language extracted from the OpaqueBehavior of Figure 2.

We use Xtext to parse and convert the extracted fragments of surface language to a format that is readable by Xtend. Because Xtext uses ANTLR,

```
Void addReadVariableAction(
   uml::Activity a, uml::Package p,
surfacelanguage::Variable v,
   uml::InputPin ip
) :
   let act = new uml::ReadVariableAction :
   let op = new uml::OutputPin :
   let of = new uml::ObjectFlow :
   a.node.add(act)
-> a.edge.add(of)
-> act.setResult(op)
-> act.setVariable(v.createVariable(p))
-> of.setSource(op)
-> of.setTarget(ip)
;
```

Figure 8. An Xtend extension that adds a ReadVariableAction to an Activity

the class of textual representations that can be parsed is restricted to those that can be described by an $LL(k)$ grammar. A disadvantage of using Xtext is that we had to modify our grammar for this reason.

One of the advantages of using the tools offered by the openArchitectureWare platform is their portability. The platform is a collection of plug-ins for Eclipse, and both Eclipse and these plug-ins are available on a number of different operating systems.

Figure 8 shows a part of the transformation implemented in Xtend from behaviour modelled using our surface language to Activities. The figure shows that a new ReadVariableAction, an OutputPin and an ObjectFlow are created, by defining local variables using *let expressions*. These expressions are followed by a *chain expression*, which denotes the sequential evaluation of the expressions connected by the "`->`" symbols. The last two of these expressions use the ObjectFlow to connect the OutputPin of the ReadVariableAction to the InputPin of another Action.

6 Other Applications of our Approach

Our approach is not only suitable for the embedding of our textual surface language in Activities. The concept of OpaqueBehavior described in Section 3.2 can, for instance, also be used to embed textual languages describing behaviour in other parts of the UML. Similar concepts, like OpaqueExpression and OpaqueAction, can be used to embed textual languages for other purposes than describing behaviour. It is possible, for instance, to use a subset of Java as an expression language for UML StateMachines.

Thus far, we described how UML models combined with our surface language can be transformed to equivalent UML models. The result of the trans-

formation described in Section 3.3, however, is only defined if the names used in the fragments of surface language of an input model correspond with elements that exist in the rest of the model. To check whether models meet this condition, we have implemented another version of our transformation, which performs a simple form of checking. This transformation takes a UML model containing fragments of surface language as input and transforms this into a list of error messages. The transformation traverses the model and the fragments of surface language, and checks whether the names used in the statements of the surface language correspond to elements that exist in other parts of the model. If the behaviour shown in the note in Figure 1 would refer to an attribute $self.b$, for instance, the transformation would produce a message stating that class A does not have an attribute named b.

7 Related Work

We chose to design and implement a new surface language, instead of implementing a design proposed by others. Section 7.1 describes two existing proposals for surface languages and indicates why we decided not to implement either of them.

There are many alternatives for the languages we used to implement our surface language. Section 7.2 lists a number of alternatives for the grammarware we used and Section 7.3 lists a number of alternatives for the modelware we used.

Section 7.4 describes another approach for integrating textual and graphical modelling languages.

7.1 Surface Languages

Dinh-Trong, Ghosh and France propose an Action Language based on the syntax of Java [2]. We decided not to implement their Action Language because their definition of the language contains a number of primitive types and Java constructs whose relation to the UML is not specified. Other important features of their language are that parameters that serve as input or output of an Activity and attributes with multiplicity greater than one are not taken into account.

Haustein and Pleumann propose a surface language that is an extension of the OCL [4,10]. They embed OCL expressions in their language by adding an Action to the UML that evaluates an OCL expression and returns the resulting value. We took a different approach, because we wanted to design and implement a simple alternative for activity diagrams that did not rely on or incorporate other languages. Incorporating an expression language like the OCL in our language would introduce a large number of language constructs that have no relation to our primary interest, which is the specification of behaviour.

7.2 Grammarware

SDF is based on SGLR, a scannerless generalized LR parser [22]. As an alternative to using SDF, SGLR can be used to parse textual representations of models. Since SGLR can parse arbitrary languages with a context-free syntax and context-free languages are closed under union, multiple syntax definitions can be combined into one without any modifications to the original syntax definitions, as is the case for SDF.

Other common tools used for parsing, such as ANTLR, JavaCC [19] and YACC [5], can also be used to parse textual representations of models. They pose more restrictions on the grammars used for the description of the textual representations, however, since the grammars need to be of the *LALR* or the *LL* class.

After parsing the textual representations of models, the resulting parse trees have to be transformed. Besides using special purpose transformation tools, generic programming languages can be used to manipulate the parse trees. The source transformation language TXL [1] is an example of a special purpose language. Paige and Radjenovic [14], and Liang and Dingel [9] have experimented with TXL in the context of model transformation. Although their research also deals with using grammarware for transformations related to models, it differs from ours because it does not focus on the integration of text-based and metamodel-based languages.

7.3 Modelware

TCS [7] is an alternative for Xtext. It is suited for both text-to-model and model-to-text transformations, and uses one specification to define the transformations in both directions. In the case of TCS, the main constructs are called templates. These templates are similar to the rules of Xtext; each template specifies the textual representation of an instance of an element of the metamodel.

Figure 9 illustrates how the resulting languages differ from our surface language in case a straightforward mapping, like those offered by Xtext and TCS, is used without additional transformations. The behaviour shown in Figure 9 is equivalent to the behaviour shown in Figure 7. The description of behaviour shown in Figure 9 is much more wordy than that of Figure 7, even for such a trivial example.

There are many languages for model transformation, including QVT [13], ATL [6] and Epsilon [8]. Since our approach does not rely on any specific properties of Xtend, each of these transformation languages can replace Xtend in our implementation.

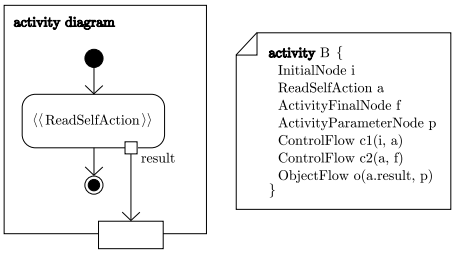


Figure 9. An activity diagram and a straightforward textual equivalent

7.4 Embedding Textual Modelling into Graphical Modelling

Scheidgen's approach for integrating textual and graphical modelling languages [18] is based on the fact that Eclipse uses the *Model View Controller* pattern [17]. A mapping from textual notation to metamodel elements is used to generate a model from a textual representation of that model and vice versa.

This custom textual notation and the graphical notations provided by Eclipse provide independent *Views* for the same *Model*. The *Controller* is used to modify the underlying model, without interfering directly with the other views.

The embedded text editor contained in the implementation of this approach offers syntax highlighting and code completion. Similar to Xtext and TCS, the language describing mappings from textual notation to metamodel elements offers only straightforward mappings.

8 Conclusions and Future Work

We investigated two approaches for the integration of textual and graphical modelling languages, by implementing a textual surface language as an alternative for activity diagrams. We described this surface language, the two approaches, the implementations that follow these approaches, and a number of related applications.

The approach using grammarware transforms models containing fragments of surface language to plain models by rewriting the XMI representations of these models. A downside of this approach is that dealing with the XMI representation of models lowers the level of abstraction of these transformations. The current implementation can only parse one variant of XMI, but a future extension that introduces an intermediate language poses a solution for this shortcoming.

The approach using modelware extracts fragments of surface language from a model, converts these fragments to a representation based on metamodels, transforms them to equivalent Activities, and replaces the original fragments with the equivalent Activities. An advantage of this approach is that all of these operations can be performed from within one modelling environment. A disadvantage of the current implementation of this approach is that the available tools pose more restrictions on the grammar of the language we embed, in comparison to the approach using grammarware. Investigating the use of more advanced parsing technology as a basis for these tools is another promising direction for future research.

The approaches we presented are not limited to the transformation of models to equivalent models. We also implemented a transformation that transforms models containing fragments of surface language into a list of error messages, thus providing a simple form of checking.

Our approaches provide advantages over the approaches described in Section 7, because they both offer a more complex mapping from textual representations to metamodel elements, which can be used to obtain simpler textual representations. The fact that the implementation using grammarware poses less restrictions on the syntax of the textual language is also an advantage over these approaches.

References

[1] Cordy, J. R., *The TXL source transformation language*, Science of Computer Programming **61** (2006), pp. 190–210.

[2] Dinh-Trong, T., S. Ghosh and R. France, *JAL: Java like Action Language, version 1.1* (2006), Department of Computer Science, Colorado State University.

[3] Efftinge, S. and M. Völter, *oAW xText: A framework for textual DSLs*, in: *Workshop on Modeling Symposium at Eclipse Summit*, 2006.

[4] Haustein, S. and J. Pleumann, *OCL as Expression Language in an Action Semantics Surface Language*, in: O. Patrascoiu, editor, *OCL and Model Driven Engineering, UML 2004 Conference Workshop, October 12, 2004, Lisbon, Portugal* (2004), pp. 99–113.

[5] Johnson, S. C., *Yacc: Yet another compiler compiler*, in: *UNIX Programmer's Manual*, Holt, Rinehart, and Winston, 1979 pp. 353–387.

[6] Jouault, F., F. Allilaire, J. Bézivin, I. Kurtev and P. Valduriez, *ATL: a QVT-like transformation language*, in: *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications* (2006), pp. 719–720.

[7] Jouault, F., J. Bézivin and I. Kurtev, *TCS: a DSL for the specification of textual concrete syntaxes in model engineering*, in: *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering* (2006), pp. 249–254.

[8] Kolovos, D. S., R. F. Paige and F. Polack, *The epsilon transformation language*, in: A. Vallecillo, J. Gray and A. Pierantonio, editors, *ICMT*, Lecture Notes in Computer Science **5063** (2008), pp. 46–60.

[9] Liang, H. and J. Dingel, *A Practical Evaluation of Using TXL for Model Transformation*, in: *SLE '08: Proceedings of the 1st International Conference on Software Language Engineering*, 2008.

[10] Object Management Group, *Object Constraint Language, OMG Available Specification, Version 2.0* (2006).

[11] Object Management Group, *MOF 2.0/XMI Mapping, Version 2.1.1* (2007).

[12] Object Management Group, *Unified Modeling Language: Superstructure 2.1.2* (2007).

[13] Object Management Group, *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.0* (2008).

[14] Paige, R. and A. Radjenovic, *Towards Model Transformation with TXL*, in: *Proceedings of the workshop on Metamodelling for MDA*, 2003.

[15] Parr, T. J. and R. W. Quong, *ANTLR: A predicated-LL(k) parser generator*, Software — Practice and Experience **25** (1995), pp. 789–810.

[16] Petre, M., *Why looking isn't always seeing: readership skills and graphical programming*, Communications of the ACM **38** (1995), pp. 33–44.

[17] Reenskaug, T., *Models - views - controllers*, Technical report, Xerox Parc (1979).

[18] Scheidgen, M., *Textual Modelling Embedded into Graphical Modelling*, in: I. Schieferdecker and A. Hartman, editors, *Proceedings of the 4th European Conference on Model Driven Architecture* (2008), pp. 153–168.

[19] Sriram Sankar and Sreenivasa Viswanadha and Rob Duncan, *JavaCC: The Java Compiler Compiler* (2008), https://javacc.dev.java.net/.

[20] van den Brand, M., J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser and J. Visser, *The ASF+SDF Meta-Environment: a Component-Based Language Development Environment*, in: *Proceedings of Compiler Construction 2001 (CC 2001)*, LNCS (2001), pp. 365–370.

[21] van Deursen, A., *An overview of ASF+SDF*, in: A. van Deursen, J. Heering and P. Klint, editors, *Language Prototyping: An Algebraic Specification Approach*, World Scientific Publishing Co., 1996 pp. 1–30.

[22] Visser, E., "Syntax Definition for Language Prototyping," Ph.D. thesis, University of Amsterdam (1997).

[23] Völter, M., *openArchitectureWare 4.2 - the flexible open source tool platform for model-driven software development* (2007).

Domain-Specific Languages for Composable Editor Plugins

Lennart C. L. Kats[*,1,2] Karl T. Kalleberg[+,3] Eelco Visser[*,1,4]

[*] *Department of Software Technology*
Delft University of Technology
Delft, The Netherlands

[+] *Department of Computer Science*
University of Bergen
Bergen, Norway

Abstract

Modern IDEs increase developer productivity by incorporating many different kinds of editor services. These can be purely syntactic, such as syntax highlighting, code folding, and an outline for navigation; or they can be based on the language semantics, such as in-line type error reporting and resolving identifier declarations. Building all these services from scratch requires both the extensive knowledge of the sometimes complicated and highly interdependent APIs and extension mechanisms of an IDE framework, and an in-depth understanding of the structure and semantics of the targeted language. This paper describes SPOOFAX/IMP, a meta-tooling suite that provides high-level domain-specific languages for describing editor services, relieving editor developers from much of the framework-specific programming. Editor services are defined as composable modules of rules coupled to a modular SDF grammar. The composability provided by the SGLR parser and the declaratively defined services allows embedded languages and language extensions to be easily formulated as additional rules extending an existing language definition. The service definitions are used to generate Eclipse editor plugins. We discuss two examples: an editor plugin for WebDSL, a domain-specific language for web applications, and the embedding of WebDSL in Stratego, used for expressing the semantic rules of WebDSL.

1 Introduction

Integrated development environments (IDEs) increase developer productivity by providing a rich user interface and tool support specialized for editing code in a particular software language. IDEs enhance readability through syntax highlighting and code folding, and navigability through cross-references and an outline view.

[1] This research was supported by NWO projects 612.063.512, *TFA: Transformations for Abstractions*, and 638.001.610, *MoDSE: Model-Driven Software Evolution*.

[2] Email: `l.c.l.kats@tudelft.nl`

[3] Email: `karltk@ii.uib.no`

[4] Email: `visser@acm.org`

Rather than providing an extensive programming environment for only one, specific language, modern IDEs such as Eclipse provide an extensible platform for integrating language processing tools for multiple languages using a plugin architecture. The flexibility of these platforms supports the integration of tools for specific languages such as editors and compilers, as well as language-independent tools, such as version control and build management systems.

Despite the extensiblity of IDE platforms, implementing state-of-the-art support for a new language is a daunting undertaking, requiring extensive knowledge of the sometimes complicated and highly interdependent APIs and extension mechanisms of an IDE framework, and an in-depth understanding of the structure and semantics of the subject language. The popular Eclipse IDE provides a cross-platform, highly extensible software platform, offering support for various languages as well as language-independent utilities. Eclipse is primarily written in Java and offers topnotch Java development tools (JDT) for the Java language. Unfortunately, Eclipse offers little opportunity for reuse of the JDT components to support other languages. Furthermore, because of its scale and broad applicability, the standard extension interface and framework are rather complex.

Key to the efficient development of IDE components are *abstraction*, to eliminate the accidental complexity of IDE frameworks, *modularity*, to reuse definitions of editor components, and *extensibility*, to customize editor components. In the design of abstractions to increase expressiveness, care must be taken to avoid a prohibitive loss of coverage, i.e. the flexibility to customize aspects of a component. Reuse is particularly important when considering *language extension* and *embedding*, as occurs for example in scenarios such as domain-specific language embedding [3,1], and meta-programming with concrete object syntax [3]. Ideally, language combinations are defined modularly, and likewise their IDE support should be composed from IDE support for the constituent languages.

Language development environments facilitate efficient development of languages processing tools. While in recent years a considerable number of language development environments have been developed [5,7,11,13,14,17,18], they offer limited support for modular and reusable editor definitions, with the exception of MontiCore [13,14] and the Meta-Environment [17,18].

MontiCore offers a certain degree of support for language extensions by merging productions of constituent grammars. However, since MontiCore's parser is based on the LL(k) formalism, extensions share the same scanner. Thus, adding extensions with a different lexical syntax is not possible. Other extensions may cause incompatibilities with existing sources (just as the introduction of the `assert` keyword in Java 1.4 excluded programs using `assert` as identifiers, which used to be valid Java). MontiCore also provides an alternative approach, dynamically switching to different scanners and parsers for blocks of code, but this is restricted to embedded languages. In contrast, the Meta-Environment uses Scannerless Generalized LR (SGLR) parsing, which is closed under composition and supports fine grained extensions and embeddings. SGLR has been applied to Java, C, PHP, as well as embeddings and extensions based on these languages [3]. The Meta-

Environment has extensive tool support for creating, debugging and visualizing grammars parsed by SGLR. However, it only offers very limited support for the "standard" editor services programmers expect, such as interactive syntax highlighting, reference resolving, or integration of utilities such as version control.

In this paper, we describe SPOOFAX/IMP, a system that integrates a Java-based implementation of SGLR into the Eclipse environment. To adequately cope with the complexity involved in IDE development *and* evolution, SPOOFAX/IMP uses compositional *editor service descriptors*, defined in domain-specific languages (DSLs), for the specification of IDE components. The DSLs provide a concise notation for specific types of syntactic and semantic editor services. Due to their declarative nature, descriptors can be composed, supporting a modular definition of both stand-alone and composite subject languages. From the descriptors, SPOOFAX/IMP generates an Eclipse plugin, which can be dynamically loaded into the same workspace as the descriptors are developed in — ensuring a shorter development cycle than the customary approach of starting a secondary Eclipse instance.

To further assist in efficient IDE development, SPOOFAX/IMP automatically derives syntactic editor service descriptors from a grammar by heuristically analyzing the grammar. Using the compositional nature of the descriptors, generated service descriptors can be composed with handwritten specifications, customizing the default behaviour. This technique allows for rapid prototyping of editors, and helps in maintaining an editor as a language evolves. When a language definition evolves, the generated components of an editor can simply be re-generated. We have applied these techniques to WebDSL, a mature domain-specific language [22,9] for web application development, which serves as running example in this paper.

The implementation of SPOOFAX/IMP is based on the IMP framework [5], which is designed to be interoperable with different parsers and other tooling, making it usable for our system. Furthermore, SPOOFAX/IMP uses the Stratego language [2] for describing the (static) semantic rules of a language, used for editor services such as cross-referencing and error reporting. For interoperability with the IMP framework, we applied a variation of the program object model adapter (POM) technique, previously used to integrate Stratego and an open compiler [10]. To support IDE feedback in combination with tree transformations, we apply a form of origin tracking [21] by adapting Stratego's generic tree traversal operators to track origins during rewrites.

This paper is organized as follows. First, we describe the definition of editor services, discussing services that handle the presentation of a language in Section 2 and those based on the language semantics in Section 3. In Section 4 we discuss composition of editor services. Section 5 provides an overview of the Eclipse and IMP architecture, and how our implementation augments it. Finally, in Section 6 we discuss related work, and offer concluding remarks and directions for future work in Section 7.

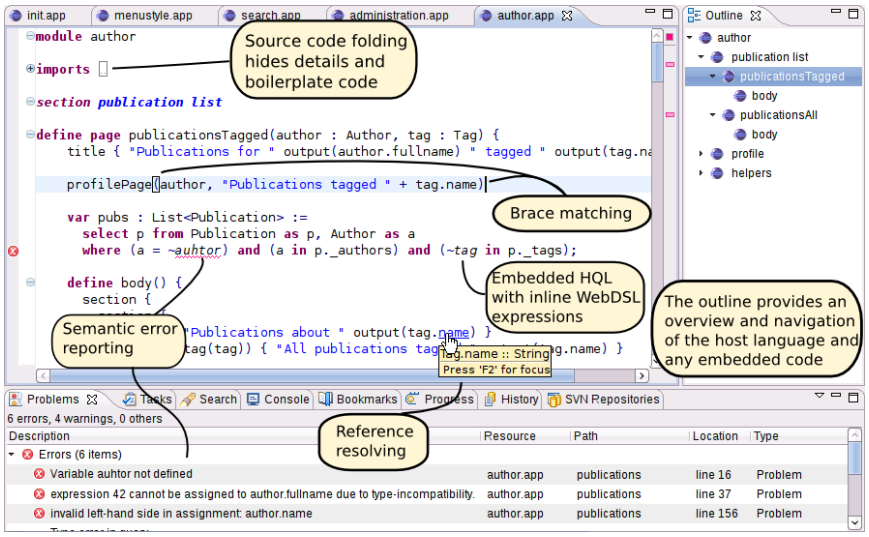


Fig. 1. Editor services for the WebDSL language.

2 Syntactic Editor Services

An editor dedicated to a particular language offers a presentation of source code specifically tailored to that language. For the most part, the presentation is directed by *syntactic editor services*. These include syntax highlighting, code folding, and the outline view. Figure 1 illustrates these and other services for WebDSL, a domain-specific language for web applications [22,9].

The grammar of a language forms the basis for the specification of syntactic editor services. For example, syntax highlighting can be specified as rules matching against grammar productions and keywords. At run-time, the grammar is used to parse the file and apply the editor service according to its grammatical structure. We employ an SGLR parser and use the modular syntax definition formalism SDF to specify the grammar of a language. Figure 2 shows part of the SDF definition for the WebDSL language. Basic SDF productions take the form

$p_1 \ldots p_n$ `->` s

which specifies that a sequence of strings matching the symbols p_1 to p_n matches the symbol s. Productions can optionally be labeled with custom names using the `{cons(`$name$`)}` annotation. SDF supports lexical and context-free productions in a single specification. The specification may be organized into modules. Languages can be composed by means of imports. In our example, WebDSL imports a separate definition for access control and reuses the Hibernate Query Language (HQL) for embedded queries. To avoid any conflicts in symbol naming, the HQL language symbols (defined by others) are renamed in the context of this grammar using the suffix `[[HQL]]`. We use this in the last two productions in Figure 2 to combine the two languages: HQL expressions can be used directly as WebDSL expressions, while WebDSL expressions in HQL are prefixed with a tilde.

```
module WebDSL

imports MixHQL[HQL] AccessControl ...

exports
  context-free start-symbols
    Start
  lexical syntax
    [a-zA-Z][a-zA-Z0-9\_]* -> Id
    ...
  context-free syntax
    "module" Id Section*              -> Start {cons("Module")}
    "section" SectionName Def*        -> Section {cons("Section")}
    "define" Mod* Id "{" Element* "}" -> Def {cons("SimpleDef")}
    ...
    Exp[[HQL]]                        -> Exp {cons("ToHQL")}
    "~" Exp                           -> Exp[[HQL]] {cons("FromHQL")}
```

Fig. 2. SDF syntax rules for the WebDSL language.

```
module WebDSL

imports
  WebDSL-Syntax   WebDSL-Colorer    WebDSL-Folding     WebDSL-Outliner
  WebDSL-Analysis WebDSL-References WebDSL-Occurrences

language description and parsing
  name           : WebDSL
  id             : org.strategoxt.imp.generated.webdsl
  description    : "Spoofax/IMP-generated editor for the WebDSL language"
  extensions     : app
  table          : include/WebDSL.tbl
  start symbols  : Unit
  url            : http://www.webdsl.org/
```

Fig. 3. The default main editor service descriptor for WebDSL.

2.1 Editor Services Composition

The main editor service descriptor module imports all service descriptors for a given language. Figure 3 shows the main module for the WebDSL language. This module can be automatically generated by specifying a language name, the SDF grammar and start production, and the file extensions of the language. Once generated, it may be customized by the developer (in our example, we added the address of the WebDSL website). Similarly, default files for the other services are generated and imported into the main module.

Each type of service, e.g. outline, folding, coloring, is defined by its own declarative description language. All editor service descriptor languages share the notion of section headings for structuring the module, such as the `language` section in this example. These specify the kind of editor service described and can optionally contain human-readable text describing the part of the file.

Since the grammar is declaratively specified using SDF, we can programmatically analyze its structure. Using a set of heuristics, our system can derive default syntactic editor service descriptors for any given grammar. For example, based on the tree structure of the grammar of Figure 2, any `Module` node will be displayed in the outline, as it contains both a lexical string and a list of subnodes. Derived rules

are placed in files clearly marked "generated" (e.g., `WebDSL-Colorer.generated`), so as to avoid confusion with handwritten code. Of course, deriving editor services based on a set of heuristic rules is never perfect, and may not suit a particular language or one's taste. Still, derived services form a good starting point for creating a new editor. Firstly, they can form the basis for a customized editor service. In particular, using editor service composition, generated editor services can be reused and customized as desired. Secondly, they quickly explain the syntax of the editor service descriptors, by means of generated, inline documentation, and by showing relevant examples that apply to the target language.

2.2 *Syntax Highlighting*

The most basic and perhaps one of the most essential editor services provided by an IDE is syntax highlighting. In Spoofax/IMP, syntax highlighting is based on the grammar of the language, rather than just a list of keywords. Thus, we correctly highlight tokens that are only considered keywords in parts of a composed grammar (such as `from`, which is a keyword in HQL but not in WebDSL). Using the grammar also enables us to report syntax errors inline.

Spoofax/IMP derives a default syntax highlighting service, based on the lexical patterns of a grammar. Figure 4 shows (fragments from) the default coloring specification for WebDSL. Each rule matches based on a lexical *token kind*, such as "keyword" for literals in the grammar, and "string" for lexical patterns with spaces. However, since SDF is scannerless, lexical tokens and productions are treated uniformly—they are just symbols. Therefore, rules can also match on productions or production types. Each rule specifies an (optional) human-readable description for display in the IDE's preference menus, together with the default color and formatting that should be applied to matching nodes.

For WebDSL, we were mostly satisfied with the default coloring rules, but wanted to emphasize sections and WebDSL expressions in HQL fragments. Therefore, we added the following rules to the `WebDSL-Colorer` module:

```
coloring customization of the default colorer
  Section._             : _ bold italic
  environment _.FromHQL : _ italic
```

The first rule specifies that terminal symbols for any production of type `Section` should be bold and italic, and may use the default color (`_`) specified elsewhere in the descriptor. The second rule is an *environment rule*, and specifies that *all* nodes below a `FromHQL` node must be displayed in italics (such as `~author` in Figure 1).

2.3 *Code Folding and Outline Display*

The *outline view* provides a structural overview of a program and allows for quick navigation. Similarly, *code folding* uses the structure of a program to selectively hide code fragments for readability. Figure 5 shows an example folding definition. The specification imports the generated folding file, adding three new rules. The last rule specifies that all import declarations should be folded by default.

```
module WebDSL-Colorer.generated
// ...documentation...
colorer default highlighting rules
  keyword : "Keywords" = magenta bold
  string  : "Strings"  = blue
  number  : "Numbers"  = darkgreen
  ...
colorer system colors
  darkred = 128 0 0
```

Fig. 4. The default WebDSL colorer.

```
module WebDSL-Folding
imports WebDSL-Folding.generated
folding additions
  Section._
  Def.SimpleDef
  Def.Imports (folded)
```

Fig. 5. Folding rules for WebDSL.

```
module WebDSL-Syntax.generated
// ...documentation...
language default syntax properties
  line comment  : "//"
  block comment : "/*" * "*/"
  fences        : [ ] ( ) { } |[ ]|
```

Fig. 6. WebDSL syntax properties.

```
[ Application -- V[ H [ "application"] _1] _2],
  Section     -- V is=2 [ H [ "section" _1]] _2],
  SimpleDef   -- V [ V is=2 [ H ["define" _1 _2
                      "{"] _3 ]
                      "}" ],
  ...
]
```

Fig. 7. Pretty printing rules for WebDSL.

2.4 Braces, Comments, and Source Code Formatting

Brace matching and selection commenting is supported through the syntax properties service, shown in Figure 6. Following the IMP framework, these features are implemented at the character level, since they must operate regardless of malformed or unclosed comments and parentheses. Thus, the descriptor specifies which pairs of strings make up parentheses, and which form comments.

Code fragments can be formatted using the Box formatting language [20]. Box allows for very flexible formatting specifications based on nested layout boxes. It supports indentation and table-based formatting. We provide this as an editor service to format code in a subject language. Figure 7 illustrates some example formatting rules, including vertical (**V**) and horizontal (**H**) boxes. These rules format WebDSL code similar to the code in Figure 1. Using the generic pretty printer package [6], part of the Stratego/XT toolset [2], we can automatically derive pretty printers from grammars.

3 Semantic Editor Services

Semantic editor services include highlighting of semantic errors, providing type information for a selected expression, and looking up the declaration of an identifier. Since these services rely on semantic analysis, our approach aims at maximal reuse of any analysis components provided by a compiler for the targeted language.

WebDSL has been implemented using Stratego/XT [2], a program transformation language and toolset that uses SDF grammars for syntax definition. The Stratego transformation language combines term-based rewriting with strategies for specifying traversals on abstract syntax trees (ASTs). Trees in Stratego are represented as first-order terms. For example, represented as a term, the AST of the program in Figure 1 is `Module("author", [Imports(...), Section(...,...)])`.

```
module WebDSL-Analysis

analysis providers and observer
  provider webdsl-front.ctree
  provider WebDSL-pretty.pp.af
  observer: editor-analyze
```

Fig. 8. Semantic analysis bindings.

```
module WebDSL-References

references tree nodes with resolvable references
  reference Call: function-resolve function-info
  reference FieldAccess: field-resolve field-info
  ...
```

Fig. 9. The reference resolving descriptor.

3.1 *Errors, Warnings, and Info Markers*

We use Stratego specifications for expressing semantic editor services. Descriptors are used to bind the IDE to the Stratego specifications. The descriptor of Figure 8 describes the interface with the type checker. It specifies all external components used to provide the semantic services: Stratego modules compiled to the Stratego core language (`.ctree` files), and a WebDSL pretty printer (`.af` files). It also specifies the *observer* function, which is notified if there are any changes to WebDSL files. This function is implemented by a Stratego rule of the following form:

```
editor-analyze:
  (ast, path, fullpath) -> (errors, warnings, infos)
  where ...
```

This rewrite rule is given a tuple with the abstract syntax tree of the modified file, its project-relative path, and an absolute file system path; as a result it can produce lists of errors, warnings and info markers. For example, for our example of Figure 1, it returned a tuple of the form:

```
([(Var("auhtor"), "Variable auhtor not defined"), ...], [], [])
```

As the first element, the errors list contains a tuple with an error message for the misspelled "auhtor" variable, which is included in the form of a copy of the original node in the tree. This simple, but effective interface allows Stratego functions to return data and reference nodes of the abstract syntax tree. We discuss the implementation of this in Section 3.3.

3.2 *Declarations and References*

Reference resolving is a feature where an IDE can find the declaration site for an identifier, accessible in Eclipse as hyperlinks for identifiers when holding the control key (see Figure 1). Similarly, occurrence highlighting highlights all occurrences of an identifier when the cursor is over one of them. Figure 9 shows the reference resolving descriptor. For `Call` or `FieldAccess` nodes, it specifies the Stratego rules that retrieve their declaration site and a description for use in tooltips (see Figure 1). These rules can be implemented using a lookup tables of identifiers to declarations (using dynamic rules [2]), similar to the typing rules in [9].

3.3 *Implementation and Tool Integration*

The main challenge of integrating IMP, Stratego and SGLR is having all tools working on the same set of data – AST nodes and tokens – at the same time. IMP defines

standard interfaces for abstract syntax trees and lexical tokens. SGLR and Stratego represent trees as terms, using their own term interfaces. SGLR is scannerless: it parses individual characters and uses no tokenizer. Tokens in the traditional sense are not present. Reconciling these differences is done in several ways. First, we use JSGLR, our own Java implementation of SGLR, and Stratego/J, a Stratego interpreter written in Java. The parse trees built by JSGLR are converted to new objects that implement the IMP interfaces. This conversion creates an IMP AST from the JSGLR parse tree, and maps literals and terminals in the parse tree to IMP tokens. The IMP AST nodes maintain links to tokens, which in turn know their exact source location. This is essential for editor services such as reference resolving or error markers that use the source locations of nodes in the tree. For integrating IMP ASTs with Stratego, we employ a variant of the program object model (POM) adapter approach [10]; our implementation of the IMP AST interface also implements the term interface necessary for Stratego rewriting.

Program analyses are implemented as Stratego transformations, and some transformations require the ability to construct AST nodes as part of an analysis. This is provided by the POM adapter, which supports node construction through an AST node factory. When Stratego programs transform trees, care must be taken that no position information is lost. For example, in the WebDSL typechecker, identifiers are given new, unique names. This means that each identifier tree node is replaced by a new value, and its parent node is transformed to one with a new set of children. Normally, tree nodes replaced in this fashion lose their associated position information. To avoid this, we made some extensions to the POM factory design for dealing with origin tracking [21] throughout the rewriting process. The factory now has added methods for supporting the primitive term traveral operators `one`, `some`, and `all`. Origin information for a term visited by a primitive traversal operator is propagated to the result after rewriting, similar to what is described in [21]: when a rewrite rule is applied to all children of a node (e.g., using the `all` operator), the origins for the old children are propagated to the new children. For the WebDSL typechecker example, this origin tracking scheme ensures that although the typechecker replaces several nodes in the tree, the new nodes maintain references to the original locations, providing proper position information for the editor services. Together with annotation support, origin tracking has proven sufficient for correct propagation of origin information during term rewriting.

4 Composition of Editor Services

The need for composing editor services arises directly from the composition required for language embedding and language extension. Consider the WebDSL generator, which is implemented in Stratego and includes embedded fragments of WebDSL, Java, and XML used for pattern matching and code generation [9]. The bottom pane of Figure 10 shows part of this generator, with WebDSL code fragments enclosed by `|[ ]|` in the Stratego code. WebDSL identifiers are italicized, and the remaining WebDSL code has a dark background.

```
module Stratego-WebDSL

imports Stratego[Stratego] WebDSL[WebDSL] ...

colorer add a background color to quoted code fragments
  environment Term[[Stratego]].ToMetaExpr:   _         metabg
              Term[[Stratego]].ToMetaExpr:   _         white  bold
  environment Term[[Stratego]].FromMetaExpr: _         white
  environment var:                           darkcyan  white  italic
  metabg = 210 230 220
```

```
type-of: |[ x := e ]| -> SimpleSort("Void")

derive-input: |[ output(e ) ]| -> |[ outputString(e) ]| where <type-of> e => SimpleSort("String")

constraint-error: Var(x) -> Error(["Variable ", x, " not defined"]) where not(type-of) // x has no type
```

Fig. 10. A composed editor based on Stratego and WebDSL.

SDF offers a uniform mechanism for composing grammars from modules. This mechanism forms the basis for both language extension and embedding. When a given SDF module imports another module, it essentially adds all grammar productions of the imported module to its own set of productions. Multiple productions for the same non-terminal are allowed; all are kept (this may give rise to ambiguities, discussed later). In the case of language embedding, suffixing (cf. Figure 2) is used to distinguish productions for the embedded language from the host language.

SPOOFAX/IMP mirrors the SDF composition mechanism – modules and imports with suffixing – but the mechanism for resolving "overlapping" definitions is different. Figure 10 defines the composition of WebDSL editor services with those of Stratego. Stratego grammar symbols are suffixed with `[[Stratego]]`, the WebDSL symbols with `[[WebDSL]]`. The first colorer rule gives WebDSL code fragments the dark background color, i.e. the text corresponding to all nodes under a `ToMetaExpr` node should be colored. The second rule sets a white background only for nodes *directly* beneath the `ToMetaExpr` node. Because this rule is more specific (i.e., applies to fewer nodes), the `|[ ]|` parentheses in the screenshot of the bottom pane have a white background, while the enclosed code has a dark background. The last two rules apply a white background to Stratego inside WebDSL fragments.

The rules for coloring, folding and outlining are all purely declarative, and share the same composition mechanics: we give priority to rules with more specific patterns, and rules defined later in the specification. New definitions can use the `_` operator to set a property to the default as specified in other rules. Finally, definitions can be disabled using a `disable` annotation.

Not all aspects of the editor services are equally composable at present. Some pretty-printer formalisms are closed under composition [6], including the one we employ. However, our present DSLs do not expose this. The most difficult services to compose are those relating to language semantics: type checking and reference resolution. A good composition formalism for language semantics is still an open research topic. The solutions offered by SPOOFAX/IMP include the use of Stratego for specifying type checking and name resolution rules in a relatively declarative style, and the use of Java for exploiting existing language infrastructure.

5 Integration with Eclipse and IMP

The Eclipse architecture is based around the OSGi component model, where each plugin is (usually) a JAR containing Java classes, a plugin manifest, optional descriptor files, and auxiliary resources, such as images. The descriptors specify which parts of the Eclipse framework a given plugin extends, and which parts of the plugin may be extended by other plugins.

IMP [5] extends Eclipse with a high-level framework and specialized tooling for developing IDE support for new languages. Developed by IBM, it is currently being used for IBM-built DSLs, such as the X10 concurrent programming language. IMP provides wizards for quickly generating initial implementations of services in terms of the IMP framework. The usual caveat applies: any user-customizations are overwritten if the wizard is run again.

SPOOFAX/IMP uses the IMP framework for implementing editor services. As an alternative to the IMP wizards, we offer DSLs which better handle language evolution and compositionality of services. By clearly separating generated code from user-written code, we avoid any risk of overwriting existing customizations as changes are made to an evolving language, or when new services are derived. While IMP makes use of uncomposable visitors for many editor services, SPOOFAX/IMP abstracts over these with DSLs. As language extensions and embeddings are composed with the subject language, the necessary visitors are automatically provided.

The OSGi model implies distributing plugins as static JARs. During development, changes made to a plugin are, in general, not reflected in the active Eclipse instance. To avoid launching a new Eclipse instance to use a new or updated editor, we use interpretation of editor services to allow dynamic loading of services into the current instance of Eclipse. For example, for the syntax highlighting specification, SPOOFAX/IMP maintains hash tables that specify which formatting style to apply to which kind of token. This approach leads to a shorter development cycle than would be possible using code generation alone. For deployment, all required files for plugin distribution are still generated.

The implementation of dynamic loading relies on the IMP notion of language inheritance. Languages may inherit services from another language. If a particular service is not available for a language, IMP tries to use the service for the parent language. We defined a new, top-level language *DynamicRoot*, from which all SPOOFAX/IMP languages inherit. Our root language implements the standard IMP extension points by providing proxy stubs. These proxies dynamically load or update the actual editor services as required. For instance, the syntax highlighting service is implemented by a class `DynamicColorer`, which initializes and updates the proper concrete `Colorer` class on demand.

Wizards and DSLs may act synergistically. Using language inheritance, editor services provided by SPOOFAX/IMP can be overridden by custom-built, Java-based services. E.g., using a handwritten outliner service only requires the use of the IMP outliner wizard to create a class overriding the standard `DynamicRoot` service.

6 Related Work and Discussion

A large body of research on editor construction and generation exists, a significant portion of which is dedicated to structured, or syntax-directed, editors. Notable early examples include the Program Synthesizer (PG), a structured program editor, and the Synthesizer Generator [15] (SG), a generator for program editors (program synthesizers). Both SG and SPOOFAX/IMP provide composable language specifications, and therefore support language extensions and language families. In SG, languages are defined as abstract syntax with attribute grammars describing the type rules. SG is agnostic about obtaining ASTs, and provides no solution for composing concrete language syntaxes. PG imposes a strictly (classical) syntax-directed editing style: programs are built from templates filled in by the user. This style ensures syntactically valid programs, but never gained widespread acceptance.

The Meta-Environment is a framework for language development, source code analysis, and source code transformation [17,18]. It includes SDF, the ASF term rewriting language, and provides an IDE framework written in Java. Basic syntax highlighting is derived from SDF grammars. Coloring may be customized similar to the `environment` construct in Figure 10. Additionally, ASF tree-traversal may used to annotate the AST with coloring directives. ASF is also used to specify the language type rules, and may include custom error messages, presented in a window similar to Figure 1. The IDE framework provides outlining but no folding or crossreferences. The Meta-Environment is presently being integrated into Eclipse.

MontiCore [13] and openArchitectureWare [7] (oAW) are tools for generating Eclipse-based editor plugins for DSLs. Both provide EBNF-like grammar formalisms which may be composed using inheritance (MontiCore) or module imports (oAW). ANTLR parsers are generated from the grammars. In MontiCore, basic editor presentation are included as grammar properties. Syntax coloring is specified as lists of keywords to highlight. Pre-defined (Java-style) comments are supported. Folding is specified by a list of non-terminals. For semantic editor services, MontiCore grammars specify events, which may be specialized with user-defined Java classes. Embedded languages are supported in MontiCore through "external" symbols in the grammar. An inheriting grammar module can implement these external symbols. The composed grammar is parsed by dynamically switching to the respective parser and lexer of one of the constituent grammars, depending on the current state of the parser. In oAW, an EMF [4] meta-model is generated in addition to the parser. Language semantics is expressed as constraints upon this model, either using an OCL-like language, or, optionally, using Java. oAW does not support embedded languages. For SPOOFAX/IMP, we provide an interface to Stratego, based on the POM adapter approach of [10] and origin tracking [21] techniques to handle semantics. SPOOFAX/IMP supports embedded languages.

Although the abstract syntax formalisms in MontiCore and oAW are both modular, the concrete syntax is limited by ANTLR grammar composition ability — LL(k) in the case of MontiCore, LL(*) for oAW, owing to the different versions of ANTLR employed. GLR parsing is closed under composition and allows gram-

mars to be composed freely. Multiple, possibly conflicting, definitions of the same non-terminal are allowed. This leads to the common criticism of GLR parsing that the resulting parse tree may contain ambiguities. This criticism is warranted, as ambiguities in grammars are by their nature undesirable, and not always trivial to discover, especially in composed grammars. Using explicit, declarative restrictions and priority orderings on the grammar, these can be resolved [19]. Other methods, including PEGs [8], language inheritance in MontiCore, and the composite grammars of ANTLR, avoid this by forcing an ordering on the alternatives of a production – the first (or last) definition overrides the others. The implicit ordering may not be the intention of the grammar engineer, and can be hard to debug as discarded alternatives cannot be inspected. In contrast, the GLR approach readily displays these alternatives. Using parser unit tests, ambiguities following from design mistakes, omissions, or regressions, can be avoided. The few remaining – intended – ambiguities are then handled immediately after parse time, when additional context information is available. SPOOFAX/IMP deals with ambiguous parse trees using a fixed pruning strategy (pick leftmost alternative from any ambiguous subtrees), which allows it to work with grammars in development.

7 Conclusion and Future Work

Providing high-quality IDE support has become paramount to the success of a new programming language. The implementation effort required for a solid, custom-built IDE is often prohibitive, in particular for domain-specific languages, language extensions, and embedded languages, with relatively small userbases. We have presented a partial solution to this, SPOOFAX/IMP, which is a a meta-tooling suite for rapidly implementing editors for Eclipse. We show several techniques that contribute towards efficient development of IDE support. First, we provide high-level, declarative DSLs for defining editor services that abstract over the IMP meta-tooling framework, and allow reuse of previously defined services through composition. Second, SPOOFAX/IMP uses heuristics to automatically derive common editor services from the grammar, services which may be adapted and co-evolved with the grammar without any trouble related to modifying generated code. Third, we use the Stratego programming language for a high-level specification of both semantic analysis and compilation. Finally, we support dynamic loading of services into the active Eclipse environment, leading to a shorter development cycle.

Some open problems remain. We want to replace the Stratego interpreter with a compiler (targeting the JVM) to address performance concerns, while maintaining the flexible interface of primitive operations as defined by the extended term factory. Similarly, we want to optimize the JSGLR parser for use in an interactive environment. The current prototype performs poorly, and, ideally, it would support incremental parsing. However, as it runs in a background thread, this has relatively little impact on the user experience. A more problematic drawback is its lack of error recovery support: in addition to reporting parsing errors, the parser should also try to parse the remainder of the file to ensure at least partial functionality of

any editor services. The IMP basis partly provides for this by ensuring that syntax highlighting is maintained in unedited regions after an error occurs. This is an area that has not yet received much attention, but [16] shows promising results.

We previously reported on a predecessor of Spoofax/IMP [11]. We have significantly expanded support for language composition, and added support for semantic analysis and dynamic loading of editor services at runtime. In part, these changes were driven by experience from the WebDSL case studies briefly reported on in this paper. In other previous work, we have applied strategic programming in the field of attribute grammars, allowing high-level, declarative specifications of semantic analyses [12]. In the future, we want to use this as a basis for defining semantic editor services. In particular, we want to investigate the feasibility of using this for compositionality of complete syntactic, semantic, and tooling descriptions of language extensions.

References

[1] M. Bravenboer, E. Dolstra, and E. Visser. Preventing injection attacks with syntax embeddings. A host and guest language independent approach. In J. Lawall, editor, *Generative Programming and Component Engineering (GPCE 2007)*, pages 3–12. ACM, October 2007.

[2] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008. Special issue on experimental software and toolkits.

[3] M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *OOPSLA 2004*, pages 365–383. ACM Press, October 2004.

[4] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, 2004.

[5] P. Charles, R. M. Fuhrer, and S. M. Sutton, Jr. IMP: a meta-tooling platform for creating language-specific IDEs in Eclipse. In R. E. K. Stirewalt, A. Egyed, and B. Fischer, editors, *ASE 2007*, pages 485–488. ACM, 2007.

[6] M. de Jonge. A pretty-printer for every occasion. In I. Ferguson, J. Gray, and L. Scott, editors, *The International Symposium on Constructing Software Engineering Tools (CoSET2000)*, pages 68–77. University of Wollongong, Australia, 2000.

[7] S. Efftinge et al. openArchitectureWare User Guide. Version 4.3. Available from http://www.eclipse.org/gmt/oaw/doc/4.3/html/contents/, April 2008.

[8] B. Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *Proceedings of the 2002 International Conference on Functional Programming (ICFP'02)*, volume 37 of *SIGPLAN Notices*, pages 36–47. ACM, October 2002.

[9] Z. Hemel, L. C. L. Kats, and E. Visser. Code generation by model transformation. A case study in transformation modularity. In J. Gray, A. Pierantonio, and

A. Vallecillo, editors, *Theory and Practice of Model Transformations. First International Conference on Model Transformation (ICMT 2008)*, volume 5063 of *LNCS*, pages 183–198. Springer, July 2008.

[10] K. T. Kalleberg and E. Visser. Fusing a transformation language with an open compiler. In A. Sloane and A. Johnstone, editors, *Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007)*, volume 203 of *ENTCS*, pages 21–36. Elsevier, April 2008.

[11] L. C. L. Kats, K. T. Kalleberg, and E. Visser. Generating editors for embedded languages. integrating SGLR into IMP. In A. Johnstone and J. Vinju, editors, *Proceedings of the Eighth Workshop on Language Descriptions, Tools, and Applications (LDTA 2008)*, April 2008.

[12] L. C. L. Kats, A. M. Sloane, and E. Visser. Decorated attribute grammars. Attribute evaluation meets strategic programming. Submitted for publication, November 2008.

[13] H. Krahn, B. Rumpe, and S. Völkel. Efficient editor generation for compositional DSLs in Eclipse. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling*, technical report TR-38, pages 218–228. University of Jyväskylä, 2007.

[14] H. Krahn, B. Rumpe, and S. Völkel. Monticore: Modular development of textual domain specific languages. In R. Paige and B. Meyer, editors, *TOOLS EUROPE 2008*, volume 11 of *LNCS*, pages 297–315. Springer-Verlag, June 2008.

[15] T. Reps and T. Teitelbaum. The synthesizer generator. *SIGSOFT Softw. Eng. Notes*, 9(3):42–48, 1984.

[16] R. Valkering. Syntax error handling in scannerless generalized LR parsers. Master's thesis, University of Amsterdam, Aug 2007.

[17] M. G. J. van den Brand. Applications of the Asf+Sdf Meta-Environment. In R. Lämmel, J. Saraiva, and J. Visser, editors, *GTTSE*, volume 4143 of *Lecture Notes in Computer Science*, pages 278–296. Springer, 2006.

[18] M. G. J. van den Brand, M. Bruntink, G. R. Economopoulos, H. A. de Jong, P. Klint, T. Kooiker, T. van der Storm, and J. J. Vinju. Using the Meta-Environment for maintenance and renovation. In *The European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 331–332. IEEE Computer Society, 2007.

[19] M. G. J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC 2002)*, volume 2304 of *LNCS*, pages 143–158. Springer-Verlag, April 2002.

[20] M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Trans. on Softw. Eng. and Methodology*, 5(1):1–41, January 1996.

[21] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15(5/6):523–545, 1993.

[22] E. Visser. WebDSL: A case study in domain-specific language engineering. In R. Lämmel, J. Visser, and J. Saraiva, editors, *International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, volume 5235 of *LNCS*, pages 291–373. Springer, October 2008.

Fundamental Nano-Patterns to Characterize and Classify Java Methods

Jeremy Singer [1] Gavin Brown Mikel Luján Adam Pocock Paraskevas Yiapanis

University of Manchester, UK

Abstract

Fundamental nano-patterns are simple, static, binary properties of Java methods, such as ObjectCreator and Recursive. We present a provisional catalogue of 17 such nano-patterns. We report statistical and information theoretic metrics to show the frequency of nano-pattern occurrence in a large corpus of open-source Java projects. We proceed to give two example case studies that demonstrate potential applications for nano-patterns. The first study involves a quantitative comparison of two popular Java benchmarking suites, in terms of their relative object-orientedness and diversity. The second study involves applying machine learning techniques to program comprehension, using method nano-patterns as learning features. In both studies, nano-patterns provide concise summaries of Java methods to enable efficient and effective analysis.

1 Introduction

Imagine you see the fragment of Java source code given in Figure 1, and you have the task of describing this method concisely to another software developer. How would you achieve this goal?

In this paper, we advocate the use of *nano-patterns* to characterise Java methods. Nano-patterns are properties of methods that are:

- *simple*: They can be detected by manual inspection from a Java developer, or by a trivial automated analysis tool.
- *static*: They should be determined by analysis of the bytecode, without any program execution context.
- *binary*: Each property is either true or false for a given method.

For instance, from our current set of 17 nano-patterns, the `fib` method in Figure 1 exhibits only two nano-patterns: namely Recursive and LocalReader.

[1] Email: jsinger@cs.man.ac.uk

Note that information is also conveyed by the fact that certain patterns are *not* exhibited: examples include ObjectCreator and Looping.

1.1 Patterns

At the high level, *design patterns* [11] encapsulate developer practice, whether that be existing conventions or managerial aspirations for better practice. These design patterns are described in terms of software architecture, using technical prose or UML diagrams. Such patterns describe re-usable templates for structuring software. Due to their high level nature, they are not directly executable or verifiable.

Recently there has been much interest in *automatic* detection of *low level patterns*, particularly in static analysis of Java bytecode. Gil and Maman [12] introduce the concept of *micro patterns* to characterize Java classes. They propose the formulation of *nano-patterns* to characterize methods within Java classes, however they do not elaborate on this idea. Høst and Østvold [15] provide a set of simple Java method attributes, which we term *fundamental nano-patterns*. In this paper, we extend Høst and Østvold's attribute set to give a fuller catalogue of fundamental nano-patterns. These patterns encapsulate Java language-specific idioms that are the *lingua franca* for experienced software developers. It must be emphasized that this catalogue is still *provisional*; we anticipate adding new nano-patterns over time.

There are many potential applications for these kinds of low level patterns. The list below mentions a number of applications that have been the subject of recent research investigations.

(i) Catalogues of idioms to enable novice developers to gain experience at reading and writing code [15].

(ii) Tools to detect bugs from anomalies in pattern usage and interactions [16,19].

(iii) Auto-completion hints in development environments [19].

(iv) Succinct characterization of code [3].

(v) Empirical evaluation coding styles and standards in a common framework [12].

```
int fib(int x) {
   if (x<=1)
     return 1;
   else
     return fib(x-1) + fib(x-2);
}
```

Fig. 1. Fragment of Java source code to be characterized concisely

(vi) Relating dynamic program behaviour with patterns, to guide just-in-time optimization decisions [17].

1.2 Contributions

The key contributions of this paper are:

(i) A categorized catalogue of fundamental nano-patterns, each with a clear definition that would enable simple mechanical detection of the pattern from bytecode, Section 2.

(ii) Formal evaluations of the nano-pattern catalogue, using information theory (Section 3) and data mining (Section 4) techniques.

(iii) Two case studies that demonstrate how nano-patterns can be used to compare different code bases (Section 5) or to aid program comprehension via large-scale statistical analysis of Java methods (Section 6).

2 Nano-Pattern Catalogue

Nano-patterns are simple properties exhibited by Java methods. They are *traceable*; that is, 'they can be expressed as a simple formal condition on the attributes, types, name and body' of a Java method [12]. They should be automatically recognisable by a trivial static analysis of Java bytecode.

Høst and Østvold [15] present a catalogue of *traceable attributes* for Java methods. They argue that these attributes could be used as building blocks for defining nano-patterns. In this paper, we refer to these traceable attributes as *fundamental nano-patterns*, which could potentially be combined to make *composite nano-patterns*.

We have supplemented Høst and Østvold's original catalogue of fundamental nano-patterns [15]. The full set of our fundamental nano-patterns is given in Table 1. The original patterns are given in plain typeface, and our new patterns are given in bold typeface. Another novelty is that we have grouped these patterns into four intuitive categories.

It is easy to see how *composite* nano-patterns could be constructed from logical combinations of *fundamental* nano-patterns. For instance, the PureMethod nano-pattern might be specified as:

$\neg$ FieldWriter $\wedge$ $\neg$ ArrayWriter $\wedge$ $\neg$ ObjectCreator $\wedge$ $\neg$ ArrayCreator $\wedge$ Leaf

A more complex definition of method purity would remove the leaf method restriction, and replace it with the recursive constraint that all method calls must also be pure methods. However this definition would require whole-program analysis, which is considered non-trivial and therefore not suitable for a nano-pattern. Note that in the remainder of this paper, we restrict attention to *fundamental* nano-patterns only.

category	*name*	*description*
Calling	NoParams	takes no arguments
	NoReturn	returns `void`
	Recursive	calls itself recursively
	SameName	calls another method with the same name
	Leaf	does not issue any method calls
Object-Orientation	ObjectCreator	creates `new` objects
	FieldReader	reads (static or instance) field values from an object
	FieldWriter	writes values to (static or instance) field of an object
	TypeManipulator	uses type casts or instanceof operations
Control Flow	**StraightLine**	no branches in method body
	Looping	one or more control flow loops in method body
	Exceptions	may throw an unhandled exception
Data Flow	**LocalReader**	reads values of local variables on stack frame
	LocalWriter	writes values of local variables on stack frame
	ArrayCreator	creates a new array
	ArrayReader	reads values from an array
	ArrayWriter	writes values to an array

Table 1
Catalogue of fundamental nano patterns. Boldface names are for original patterns we have devised, all other patterns come from Høst and Østvold's catalogue.

2.1 Detection Tool

We have developed a command line tool to detect nano-patterns for methods in Java bytecode class files, based on the ASM bytecode analysis toolkit [6]. Our tool reads in a class file name specified as a command line argument, and dumps out a bitstring of nano-patterns exhibited for each method in the class. The detection tool is written in Java; it is only 600 source lines of code. Our code makes extensive use of data structures and visitor code from the ASM API. The tool operates in two different ways to detect specific nano-patterns:

(i) Some patterns are found by simple iteration over a method bytecode array, searching for specific bytecode instructions that indicate particular nano-patterns. For example, the `newarray` bytecode indicates the ArrayCreator nano-pattern.

(ii) Other patterns are found by simple regular expression matches on method signatures. For example, if the method type signature contains the string `()` then the method exhibits the NoParams nano-pattern.

program	version	description
Ashes Suite	1st public release	Java compiler test programs
DaCapo	2006-10-MR2	Object-oriented benchmark suite
JBoss	3.2.2	Application server
JEdit	4.3	Java text editor application
JHotDraw	709	Java graphics application
Jikes RVM	2.9.1	Java virtual machine, includes classpath library
JOlden	initial release	Pointer-intensive benchmark suite
JUnit	4.4	Test harness
SPECjbb	2005	Java business benchmark
SPECjvm	1998	Simple Java client benchmark suite

Table 2
Java benchmarks used in nano-pattern coverage study

We envisage that it should be possible to automate the generation of ASM-based detection code for specific nano-patterns, given some kind of formal specification of the nano-pattern characteristics. A meta-language like JTL [8] may be useful here. We do not address this issue in the current research.

2.2 Statistics

We analyse a large and varied corpus of Java programs; the details are given in Table 2. These are all commonly available industry-standard benchmark suites and open-source Java applications, that have been used in previous research-based Java source code case studies.

In total, there are 43,880 classes and 306,531 methods in this corpus. We run our nano-pattern detection tool on all these classes. Table 3 summarises the results. It gives the proportion of methods that exhibit each kind of nano-pattern. The *overall coverage* represents the percentage of all analysed methods that exhibit any nano-pattern. Since this score is 100%, all methods analysed exhibit at least one nano-pattern from our catalogue. The mean number of nano-patterns per method is 4.9.

3 Information Theoretic Characterization

Information theoretic entropy measures the *uncertainty* associated with a random variable. In this section, we consider our nano-pattern detector tool as a black box supplying values that represent nano-pattern bitstrings. For each of the different potential bitstrings, there is an associated probability based on its frequency of occurrence. (We estimate probabilities by frequencies in our corpus of 306,531 methods.) Given the set of all possible bitstrings B, we denote the probability of the occurrence of a particular bitstring $b \in B$ as p_b.

nano-pattern	% coverage
LocalReader	89.4
StraightLine	63.6
FieldReader	51.4
Void	50.6
NoParams	39.2
SameName	32.4
LocalWriter	31.1
ObjectCreator	26.5
FieldWriter	26.5
Leaf	20.3
TypeManipulator	15.2
Exceptions	13.6
Looping	11.3
ArrayReader	6.7
ArrayCreator	5.4
ArrayWriter	5.3
Recursive	0.7
Overall	100.0

Table 3
Coverage scores for each nano-pattern on the corpus of Java programs

We compute the entropy H (after Shannon) as:

$$H = -\sum_{b \in B} p_b \log_2(p_b)$$

A low entropy score indicates low uncertainty in the underlying random variable, which means that nano-patterns are very predictable. This would reduce their utility for classification. On the other hand, a high entropy score indicates high uncertainty. The maximum entropy score is $\log_2|B|$ where $|B|$ is the number of potential bitstrings. Since there are 17 different nano-patterns in our catalogue, the maximum entropy score would be 17. This would mean all nano-patterns are independent, and have a 50% chance of being exhibited by a method.

In fact, from the 306,531 methods we measured, the entropy of the bitstrings is **8.47**. This value is relatively high, which means the nano-patterns for a method are not easily predictable. There are some inter-dependencies between patterns, but these are generally non-trivial. (The next section describes cross-pattern relationships in detail.)

4 Data Mining Characterization

4.1 Background

Data Mining is 'the nontrivial extraction of implicit, previously unknown, and potentially useful information from data' [10]. A number of techniques exist to perform data mining on large data sets. One of the most popular techniques is *association rule* mining from sets of items in a data set, introduced by [1]. Association rules are obtained via *frequent pattern mining*. Association rules take the form of logical implications. Their primary use is for *market basket analysis*, where vendors search for items that are often purchased together [5].

We are interested in sets of nano-patterns that are frequently exhibited together, by Java methods. Such association rules have the form $A \rightarrow B$, meaning that if method m exhibits nano-pattern A, then this implies m also exhibits B. Along with each rule, there are two related measures of interest: *support* and *confidence*. The support is the proportion of methods that exhibit both A and B in relation to the total number of methods analysed. The confidence is the proportion of methods that exhibit both A and B in relation to the total number of methods that exhibit A. A rule is only retained if it satisfies user-determined minimum thresholds for both support and confidence [2].

4.2 Nano-Pattern Analysis

We perform association rule mining on the complete set of 306,531 methods for which we have nano-pattern data. The rule mining algorithm produces hundreds of rules. However we immediately discard all rules involving the LocalReader nano-pattern; since it is such a prevalent pattern, any rules involving it are not really meaningful. Many rules remain after this initial pruning. Some of these are obvious, for instance: ArrayCreator implies ArrayWriter with high confidence. In the remainder of this section, we report on three interesting rules that occur due to common Java programming idioms. Each of these rules exceeds our thresholds for support and confidence. We carry out further statistical analysis using the *lift* and χ^2 measures to determine whether there are statistically significant correlations between the associated nano-patterns in each rule. In each case we find that the nano-patterns are significantly positively correlated.

(1) $$\text{Looping} \rightarrow \text{TypeManipulator}$$

This rule is caused by the prevalence of `java.util.Iterator` objects used in while loops over data structures from the Java Collections framework. The code listing below gives an outline example.

```
while (i.hasNext()) {
  Element e = (Element)i.next();
  // ...
```

```
}
```

In older versions of Java, all objects are coerced to the `Object` supertype when they are stored in library container data structures. Even with addition of generics in Java 5 source, type casts are still present in Java bytecode for retrieving objects from container data structures. Therefore this rule is an idiomatic artifact of the Java source to bytecode transformation.

(2) $\mathsf{ArrayReader} \rightarrow \mathsf{Looping}$

This rule is caused by the idiom of iterating over an entire array, reading each element. The code listing below gives an outline example.

```
for (int i=0; i<a.length; i++) {
    // ...
    doWork(a[i]);
    // ...
}
```

(3) $\mathsf{FieldWriter} \wedge \mathsf{StraightLine} \rightarrow \mathsf{NoReturn}$

This rule is due to the prevalence of object-oriented *setter* accessor methods. Such methods take a single argument, write this value to a field of the current object and return `void`. The code listing below gives an outline example. One would expect to see this kind of rule for well-written programs in any object-oriented language.

```
public void setXYZ(Foo xyz) {
    this.xyz = xyz;
    return;
}
```

4.3 Applications

There are many potential applications for these kinds of association rules. We outline three areas below.

(i) Detection of high-level design patterns from low-level nano-patterns. In general, design pattern discovery is acknowledged to be difficult [14,9]. We have shown above that some combinations of low-level features are potential indicators for higher-level patterns. Gueheneuc et al [13] explore this concept further, although with a possibly more restrictive set of static code features.

(ii) A 'Programmer's Lexicon' style guidebook for novice programmers [15], outlining common and idiomatic programming conventions. Each discovered convention requires manual annotation to provide some measure of *goodness*. In particular, it is likely that prevalent *anti-patterns* may be discovered.

(iii) Identification of potential bugs. Given a large and varied corpus of code, we can extract a set of high-confidence association rules. If these rules are not kept in new code, an online interactive checker can inform the developer of the rule violations [19].

5 Case Study A: SPECjvm98 vs DaCapo

In this section, we use nano-patterns to contrast two Java client-side benchmark suites. In general, it is difficult to quantify the differences between two sets of programs: However we demonstrate that nano-patterns provide a good basis for differentiation.

The *SPECjvm98* benchmark suite was originally intended to evaluate the performance of commercial Java virtual machine (JVM) implementations. However due to its small size and relative age, it is now only used as a target for academic research such as *points-to analysis* [20]. A potential replacement for SPECjvm98 is the *DaCapo* benchmark suite, compiled by an academic research group. The DaCapo introductory paper [4] presents an extensive empirical study to highlight the differences between these two benchmark suites. The authors claim that DaCapo is superior to SPECjvm98 for two main reasons:

(i) DaCapo programs are more object-oriented than SPECjvm98.

(ii) DaCapo programs are more diverse in their behaviour than SPECjvm98.

Using our nano-patterns catalogue, we should be able to provide new quantitative evaluations of these criteria for the two benchmark suites.

5.1 Object Orientation

The DaCapo paper [4] argues that the DaCapo suite is 'more object-oriented' than SPECjvm98. The static analysis study that backs up this claim employs Chidamber and Kemerer metrics [7]. We can evaluate the level of static object orientation in each benchmark suite, by considering the four nano-patterns that deal with object orientation. Recall from Table 1 that these are ObjectCreator, FieldReader, FieldWriter and TypeManipulator. (In this study we abbreviate these nano-patterns as OC, FR, FW and TM respectively.)

Table 4 presents the results of this analysis. For each benchmark suite, we consider every Java application separately. For each application, we perform static analysis on all methods defined in benchmark classes that are loaded by a JVM during an execution of that benchmark with the default workload. From this analysis, we report the proportion of methods that exhibit each OO nano-pattern. We also report the overall OO coverage, which gives the proportion of methods that exhibit at least one OO nano-pattern.

From these results, it is not immediately clear to see whether DaCapo is more object-oriented than SPECjvm98. They have similar overall coverage

	benchmark	# methods	% OC	% FR	% FW	% TM	% cov
SPECjvm98	_201_compress	44	13	65	52	0	86
	_202_jess	673	33	50	23	8	75
	_205_raytrace	173	16	58	40	2	86
	_209_db	34	38	79	50	32	94
	_213_javac	5601	29	61	26	10	77
	_222_mpegaudio	280	17	60	38	2	79
	_227_mtrt	177	15	57	39	2	85
	_228_jack	302	23	36	49	10	66
	geomean	249	21	57	38	5	81
DaCapo	antlr	1788	41	62	39	13	81
	bloat	2718	33	66	33	22	85
	chart	4182	33	59	26	12	82
	eclipse	5385	27	58	29	16	79
	fop	5180	24	46	32	7	76
	hsqldb	2767	21	58	22	12	72
	jython	6549	25	55	19	19	75
	luindex	963	28	56	33	9	79
	lusearch	1252	27	58	32	10	81
	pmd	4923	20	45	26	13	66
	xalan	5512	20	54	28	10	75
	geomean	3180	27	56	28	12	77

Table 4
Object-oriented nano-pattern coverage for each benchmark

scores for the OO nano-patterns, in relative terms. However note that absolutely, DaCapo is much larger than SPECjvm98. The OO metrics given in the original DaCapo paper were absolute figures too.

A higher proportion of methods create objects in DaCapo, and it also has many more type manipulating methods. These are clear indications of object orientation. On the other hand, there are similar amount of object field reading for both suites. Interestingly, SPECjvm98 seems to perform much more object field writing. We investigate the difference between accesses to static and instance fields, since FR and FW cover both static and instance accesses by definition. Again we found similar statistics in both suites: around 20% of reads are to static fields, and less than 10% of writes are to static fields.

One potential limitation of this study is that the nano-pattern catalogue does not presently capture all object-oriented behaviour. For instance, we do not have any measure of method overriding via virtual method calls. Also we make no distinction between accessing object fields through a `this` pointer and other pointers. Perhaps a richer set of nano-patterns would provide a clearer picture.

5.2 Diversity

Nano-patterns can be used to indicate similarity between methods; we assert that similar methods should exhibit similar nano-patterns. The DaCapo paper [4] criticizes the SPECjvm98 benchmarks for being overly similar. The authors take a set of architectural metrics for each benchmark and perform a principal components analysis with four dimensions. They show that the DaCapo programs are spread around this 4-d space, whereas the SPECjvm98 programs are concentrated close together.

Again, we can use nano-patterns to confirm the results of this earlier study. We consider all nano-patterns in our catalogue from Table 1. Again, we consider all methods from benchmark classes loaded during execution. To demonstrate that different benchmarks within a suite are diverse, we take two measurements for each benchmark.

(i) *Number of unique nano-pattern bitstrings*: Given a set of nano-pattern bitstrings for a single benchmark, which of these bitstrings do not appear in any other benchmark in the suite? This characterizes behaviour that is unique to one benchmark. We can count the number of such unique bitstrings as an indicator of benchmark diversity within a suite.

(ii) *Information theoretic entropy*: Given a set of nano-pattern bitstrings for each benchmark, we can compute the information theoretic entropy of that set. High entropy values indicate greater uncertainty, i.e. the bitstrings are less predictable. Again, this can indicate benchmark diversity within a suite.

Table 5 reports the results for this analysis of benchmark diversity. It is clear to see from the geometric mean scores for each benchmark suite that DaCapo benchmarks have more unique nano-pattern bitstrings per benchmark, and that the entropy of nano-pattern bitstrings is higher for DaCapo. This analysis confirms the claims in the original DaCapo paper [4] that the DaCapo suite is more diverse than SPECjvm98.

5.3 Caveats

Analysis based on nano-patterns is entirely *static*. For a true comparison between the benchmark suites (especially in relation to diversity) it would be better to look at both static and dynamic behaviour. The DaCapo study focused entirely on dynamic behaviour, whereas we have only looked at static behaviour here. However we reach the same conclusions in relation to intra-suite diversity.

On the other hand, we assert that it is still useful to perform a static comparison of the benchmark suites in isolation. Often these particular Java benchmarks are used to compare static analysis techniques (as opposed to runtime JVM performance) in which case, static object orientation and diversity become the main concern. Hence this style of empirical comparison based on

	benchmark	# methods	# unique NP sets	entropy
SPECjvm98	_201_compress	44	6	4.69
	_202_jess	673	52	6.09
	_205_raytrace	173	0	4.55
	_209_db	34	8	4.79
	_213_javac	5601	628	8.13
	_222_mpegaudio	280	32	6.48
	_227_mtrt	177	0	4.58
	_228_jack	302	24	4.92
	geomean	248.63	13.65	5.41
DaCapo	antlr	1788	28	7.22
	bloat	2718	49	7.02
	chart	4182	98	7.17
	eclipse	5385	95	8.35
	fop	5180	32	7.01
	hsqldb	2767	144	8.29
	jython	6549	136	7.13
	luindex	963	10	7.62
	lusearch	1252	13	7.65
	pmd	4923	44	7.57
	xalan	5512	110	8.08
	geomean	3179.85	50.14	7.54

Table 5
Measurements of benchmark diversity in terms of unique nano-pattern sets and nano-pattern entropy

nano-patterns is indeed valuable.

6 Case Study B: Method Clustering based on Nano-Patterns

Clustering is a form of unsupervised learning. It is used to group data points into a variable number of clusters based upon a similarity measure, usually a distance metric. This enables a quick characterisation of data into higher level groupings. In this particular context, we aim to cluster similar methods to enable program comprehension, where method similarity is based on nano-pattern bitstrings. There are two main obstacles:

(i) all our nano-pattern features are binary values, which is non-standard for clustering algorithms that generally operate on real-valued continuous data.

(ii) our nano-pattern feature space has 17 dimensions. This makes it difficult to visualize any clusterings.

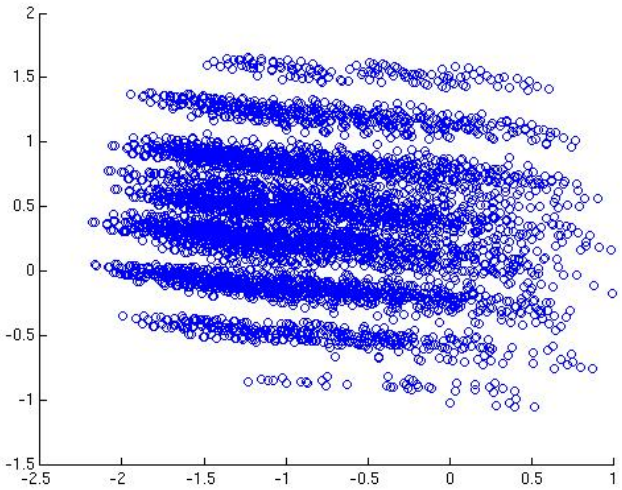


Fig. 2. 2-d projected nano-pattern data for methods in corpus (note sausage-shaped clusters)

To work around these problems, we use principal components analysis (PCA) to project our data into a continuous 2-d space. PCA transforms the data into a different space. It creates new features out of the axes of maximum variation in the original data set. This means the largest principal components contain the most information about the data. Figure 2 shows a visualization of this projected data. The first two principal components form the axes for this graph, as these account for most of the variation in the data.

The figure shows a number of different clusters, indicating that there are several groups of similar methods in the original data set. A further clustering on this data would provide a basis for relating the apparent clusters to the presence of combinations of nano-patterns in the original data set.

We note in passing that there has been previous work using clustering to analyse Java methods [18]. However our set of static method features appears to be richer than in earlier work. The application area for this analysis is mostly *program comprehension*.

7 Conclusions

In this paper, we have shown that fundamental nano-patterns can provide succinct characterizations of Java methods. We have demonstrated the capabilities of nano-patterns to provide a framework for quantitative analysis of large Java applications, and to enable learning-based techniques like data mining and clustering.

Our future work includes extending the provisional catalogue of nano-patterns. We hope to improve its object-oriented features with support for method overloading, overriding and `super()` calls. We also want to enrich our Exceptions nano-pattern to distinguish between methods that throw exceptions directly, catch exceptions, and propagate uncaught exceptions. Additional higher-level method characteristics include threading activity and use of standard Java APIs like the collections framework.

Finally, we hope to employ state-of-the-art clustering algorithms to group related methods together and analyse these results. Eventually we aim to use fundamental nano-patterns in a supervised learning context.

References

[1] Agrawal, R., T. Imielinski and A. Swami, *Mining association rules between sets of items in large databases*, in: *Proceedings of the International Conference on Management of Data*, 1993, pp. 207–216.

[2] Agrawal, R. and R. Srikant, *Fast algorithms for mining association rules*, in: *Proceedings of the 20th International Conference on Very Large Databases*, 1994, pp. 487–499.

[3] Bajracharya, S., T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi and C. Lopes, *Sourcerer: a search engine for open source code supporting structure-based search*, in: *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming, Systems, Languages, and Applications*, 2006, pp. 681–682.

[4] Blackburn, S. M. et al., *The DaCapo benchmarks: Java benchmarking development and analysis*, in: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2006, pp. 169–190.

[5] Brin, S., R. Motwani, J. Ullman and S. Tsur, *Dynamic itemset counting and implication rules for market basket data*, in: *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, 1997, pp. 255–264.

[6] Bruneton, E., R. Lenglet and T. Coupaye, *ASM: a code manipulation tool to implement adaptable systems*, in: *Adaptable and Extensible Component Systems*, 2002.

[7] Chidamber, S. and C. Kemerer, *A metrics suite for object oriented design*, IEEE Transactions on Software Engineering **20** (1994), pp. 476–493.

[8] Cohen, T., J. Y. Gil and I. Maman, *Jtl: the Java tools language*, in: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2006, pp. 89–108.

[9] Dong, J. and Y. Zhao, *Experiments on design pattern discovery*, in: *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, 2007, p. 12.

[10] Frawley, W., G. Piatetsky-Shapiro and C. Matheus, *Knowledge discovery in databases: An overview*, AI Magazine (1992), pp. 213–228.

[11] Gamma, E., R. Helm, R. Johnson and J. M. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison Wesley, 1994.

[12] Gil, Y. and I. Maman, *Micro patterns in Java code*, in: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005, pp. 97–116.

[13] Gueheneuc, Y., H. Sahraoui and F. Zaidi, *Fingerprinting design patterns*, in: *Proceedings of the 11th Working Conference on Reverse Engineering*, pp. 172–181.

[14] Heuzeroth, D., T. Holl, G. Högström and W. Löwe, *Automatic design pattern detection*, in: *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, 2003, pp. 94–103.

[15] Høst, E. W. and B. M. Østvold, *The programmer's lexicon, volume I: The verbs*, in: *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, 2007, pp. 193–202.

[16] Kim, S., K. Pan and E. Whitehead Jr, *Micro pattern evolution*, in: *Proceedings of the International Workshop on Mining Software Repositories*, 2006, pp. 40–46.

[17] Marion, S., R. Jones and C. Ryder, *Decrypting the Java gene pool: Predicting objects' lifetimes with micro-patterns*, in: *Proceedings of the International Symposium on Memory Management*, 2007, pp. 67–78.

[18] Rousidis, D. and C. Tjortjis, *Clustering data retrieved from Java source code to support software maintenance: A case study*, in: *9th European Conference on Software Maintenance and Reengineering*, 2005, pp. 276–279.

[19] Singer, J. and C. Kirkham, *Exploiting the correspondence between micro patterns and class names*, in: *Proceedings of the Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, 2008, pp. 67–76.

[20] Sridharan, M. and R. Bodík, *Refinement-based context-sensitive points-to analysis for Java*, in: *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006, pp. 387–400.

SLAMM — Automating Memory Analysis for Numerical Algorithms

John M. Dennis [1,2]

Computational and Information Systems Laboratory
National Center for Atmospheric Research
Boulder, CO 80305, USA

Elizabeth R. Jessup [3,4]

Department of Computer Science
University of Colorado
Boulder, CO 80309, USA

William M. Waite [5]

Department of Electrical and Computer Engineering
University of Colorado
Boulder, CO 80309, USA

Abstract

Memory efficiency is overtaking the number of floating-point operations as a performance determinant for numerical algorithms. Integrating memory efficiency into an algorithm from the start is made easier by computational tools that can quantify its memory traffic. The Sparse Linear Algebra Memory Model (SLAMM) is implemented by a source-to-source translator that accepts a MATLAB specification of an algorithm and adds code to predict memory traffic.

Our tests on numerous small kernels and complete implementations of algorithms for solving sparse linear systems show that SLAMM accurately predicts the amount of data loaded from the memory hierarchy to the L1 cache to within 20% error on three different compute platforms. SLAMM allows us to evaluate the memory efficiency of particular choices rapidly during the design phase of an iterative algorithm, and it provides an automated mechanism for tuning exisiting implementations. It reduces the time to perform *a priori* memory analysis from as long as several days to 20 minutes.

Key words: memory analysis, sparse linear algebra, source-to-source translation, MATLAB

1 Introduction

Traditionally, numerical algorithms have been designed to achieve the best numerical accuracy for a given number of floating-point operations [26,31,30,1,25]. However, this approach is based on the assumption that the time to perform floating-point operations dominates the overall cost of the computation. This assumption is no longer true in general because the cost of memory access has risen in comparison to the cost of floating-point arithmetic. While the performance of microprocessors has been improving at the rate of 60% a year, the performance of dynamic random access memory has been improving at a rate of only 10% [28]. In addition, advances in algorithm development mean that the total number of floating-point operations required for solution of many problems is decreasing [19]. Thus, the time to solution can no longer be considered a function of the number of floating-point operations performed alone but must rather be described as a combination of the costs of both floating-point arithmetic and memory access [19].

Creating new algorithms that demonstrate both numerical and memory efficiency [35,7,2,5,6,37,14,20,17] is a difficult task that has not been addressed extensively or in a systematic fashion. The all-too-common approach is to ignore memory efficiency until after the implementation is complete. Unfortunately, retroactive retooling of code for memory efficiency can be a daunting task. It is better to integrate memory efficiency into the algorithm from the start.

Manual analysis, which involves derivation of an analytical expression for data movement, requires extensive knowledge of computer architecture and software engineering. It is laborious, error-prone, and too complex to perform on a regular basis. Fortunately, we can automate the process by employing computational tools to quantify the memory traffic. A static analysis provides an estimate of memory use by counting variable references in the code itself; a dynamic analysis is performed as the code is run and so reflects actual data access.

One general design strategy is to prototype an algorithm in MATLAB using different memory layouts and to determine how the final version will perform by making measurements on the prototypes. Creating useful prototypes with appropriate statistics-gathering code requires specialized knowledge; this method would thus not be widely available if we didn't package

[1] The work of this author was supported through National Science Foundation Cooperative Grant NSF01, which funds the National Center for Atmospheric Research (NCAR), and through the grants: #OCI-0749206 and #OCE-0825754. Additional funding is provided through the Department of Energy, CCPP Program Grant #DE-PS02-07ER07-06.
[2] Email: dennis@ucar.edu
[3] The work of this author was supported by the National Science Foundation under grants no. CCF-0430646 and CCF 0830458.
[4] Email: Elizabeth.Jessup@Colorado.edu
[5] Email: William.Waite@Colorado.edu

```
SLAMM Memory Analysis for Body: BLGMRES
       TOTAL: Storage Requirement  Mbytes (SR) : 10.21
       TOTAL: Loaded from L2 -> L1 Mbytes (WSL): 574.64 +- 6.16
                           DGEMM    Mbytes      : 0.00 +- 0.00
                      Sparse Ops    Mbytes      : 79.50 +- 6.16
```

Fig. 1. The result of a memory analysis

that knowledge. We therefore developed the the Sparse Linear Algebra Memory Model (SLAMM) processor, a source-to-source translator that accepts MATLAB code describing the candidate algorithm and adds blocks of code to predict data movement [10].

Source-to-source translators have a long history in the development of numerical algorithms. In early 1970, Marian Gabriel reported on a translator that accepted a FORTRAN formula and produced a FORTRAN formula computing the derivative of the input formula [16]. That translator was part of a larger system of doing least-squares fits [15]. Gabriel's rationale for providing the translator echoes ours for SLAMM:

> The curve-fitting program ... requires partial derivatives of the function to be fitted, and thus the user had to supply FORTRAN expressions for these derivatives. Finding them for a complicated function is, at best, a nuisance. At worst, it is a common source of errors.

Over the years, tools have been developed to automate production of analysis and translation algorithms from declarative specifications [22]; using them, processors like SLAMM can now be constructed more easily. We used Eli [18,13], a public-domain compiler generator with sophisticated support for standard tasks such as name analysis, to generate the SLAMM processor.

Section 2 explains how the SLAMM processor analyzes a prototype and translates it to a MATLAB program. In Section 3, we summarize our experiences in applying SLAMM to algorithm development.

2 The SLAMM processor

SLAMM's goal is to predict the amount of data moved by an algorithm to be implemented in a compiled language, given a MATLAB prototype for that algorithm. Figure 1 illustrates information that SLAMM provided to the user for an algorithm named `BLGMRES`, which is a sparse linear system solver for a matrix of order 25,228:

- The sum of the storage required (SR) for all of `BLGMRES`'s variables is 10.21 Mbytes.
- The working set load (WSL) size is the amount of data load from the memory to the processors, which for `BLGMRES` is 574.64 Mbytes. This figure may be in error by up to 6.16 Mbytes, due to the difficulty SLAMM has in predicting data movement for certain programming constructs.

Table 1
SLAMM Commands

`start` *name*	Mark the beginning of a body of MATLAB code to be analyzed, and give it the name *name*.
`end` *name*	Mark the end of the body of MATLAB code named *name*.
`FuncStart` *name*	Mark the beginning of a MATLAB function to be analyzed, and give it the name *name*.
`FuncEnd` *name*	Mark the end of the MATLAB function named *name*.
`print` *name*	Request that a memory analysis be printed for *name*.
`Func` *ident*, ..., *ident*	Classify one or more identifiers as names of functions for which data movement is to be ignored.
`IncFunc` *ident*, ..., *ident*	Classify one or more identifiers as names of functions that contain significant data movement.

- There were no dense matrix-matrix operations that could be implemented as calls to the Basic Linear Algebra Subprogram (BLAS) DGEMM [12].
- Sparse matrix-vector operations account for 79.50 Mbytes of the total loads and all of the possible error.

Both static and dynamic analysis of the prototype are necessary in order to obtain this information. SLAMM performs the static analysis directly, and also produces a copy of the prototype that has been augmented with additional MATLAB code blocks. The MATLAB interpreter executes the transformed code to provide the dynamic analysis.

The SLAMM processor accepts a MATLAB prototype containing SLAMM *directives*. All SLAMM directives consist of the prefix `%SLM`, a *command*, and a terminating semicolon. A command (Table 1) is a keyword followed by one or more arguments, separated by commas.

In Section 2.1 we explain how the number of memory accesses corresponding to a variable reference in the prototype is related to the region in which it occurs. Unfortunately, not every variable reference has the same effect, as discussed in Section 2.2. The translation based upon a static analysis of the prototype is described in Section 2.3.

2.1 Regions and variable accesses

In the simplest case, each occurrence of a variable in the prototype represents an access to that variable's memory location by the final algorithm. However, the number of times the memory location is accessed depends on the execution path and the region containing the variable occurrence. As an example, consider the prototype shown in Figure 2. The conditional guarantees that

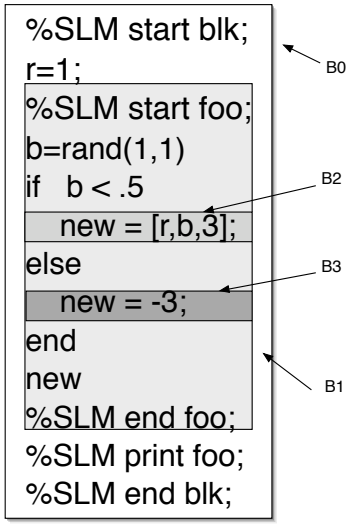


Fig. 2. Scopes for counter association

either B2 or B3, but not both, will be executed. Similarly, variables occurring in a region controlled by an iteration will be accessed many times.

SLAMM directives can also be used to delimit regions on which to perform memory analysis. In Figure 2, directives have been used to define the region B1 and give it the name `foo`. The `print` command causes an output similar to that described by Figure 1.

Name analysis is the process used to discover and record all of the relationships among identifier occurrences within program regions. This process is well understood, and can be described in terms of a few simple concepts [23]:

Range: A region of the program text. Each of the regions B0-B3 in Figure 2 constitutes a range. For each range, SLAMM calculates: the total storage requirement (SR), the amount of data loaded from memory, the working set load (WSL) size, and the amount of data stored to memory, the working set store (WSS) size.

Defining occurrence: An occurrence of an identifier that would be legal even if it were the sole occurrence of that identifier. All identifier occurrences in SLAMM are defining occurrences. For each defining occurance, SLAMM creates a new data structure by prepending `slm_` to the identifier and captures type, extent, and required memory storage.

Binding: A relationship between an identifier and a unique entity.

Scope: The region of the program text within which a binding is valid. In

SLAMM, a binding holds over the entire range containing a defining occurrence of the identifier. A defining occurrence for the same identifier in a nested range creates a new binding. Within the scope of that new binding, the old binding is invalid. The scope of the new binding therefore constitutes a "hole" in the scope of the old binding.

Name space: A collection of ranges, occurrences, bindings and scopes. SLAMM has two name spaces: the *variable* name space and the *counter* name space. Every identifier occurrence has a binding in each name space.

The only ranges that are relevant in the variable name space are those constituting files and functions. In Figure 2, the region B0 is the only range in the variable name space. The entity bound to an identifier in the variable name space represents the prototype variable named by that identifier.

All of the ranges are relevant in the counter name space. The entity bound to an identifier in the counter name space represents the number of times the prototype variable named by that identifier is accessed in the scope of that binding.

2.2 Correcting the access count

An occurrence of an identifier in the prototype does not necessarily indicate that the corresponding variable must be loaded from the memory hierarchy. Fortunately, it is possible to improve the accuracy of the base identifier counts through a series of corrections. These corrections capture properties of the translation from MATLAB to a compiled language.

An efficient implementation of the expression `n = size(A,1);`, which uses the built-in MATLAB function `size` to set the variable `n` to the row dimension of the matrix `A`, does not require access to the entire matrix `A`. The necessary functionality can be provided in a compiled language by accessing a single integer variable. The *function call correction* addresses the counting mismatch by decrementing the counts for those identifiers that occur within a function call argument list. We are careful not to decrement the count for those identifiers that occur in an expression in an argument list. For example, the function call correction does not apply to the vector identifiers `r_m1` and `r` in `m = size(r_m1'*r);`. We describe our approach for memory analysis of function calls in the next section.

The calculation of the dot product `alpha = r'*r;` shows the need for another type of correction. In this case, the identifier `r` occurs twice in a single expression. However, the vector `r` is only loaded once from the memory hierarchy. The duplicate occurrence of `r` within a single expression represents cache reuse that cannot be ignored in the memory analysis calculation. To address cache reuse, we decrement the identifier counts for any identifiers that occur multiple times within a single statement. For simplicity, this *duplicate correction* does not address the possibility of cache reuse between multiple statements.

The expression `r_m1 = r;` copies the vector `r` to vector `r_m1`. Memory copies are necessary in MATLAB for renaming purposes because MATLAB lacks a pointer construct. Single variable assignments are implemented as either a memory copy or a pointer assignment in a compiled language. We assume that, for an efficiently implemented algorithm, variables with a large storage requirement (e.g., vectors) use pointer assignments. Variables with a small storage requirement, such as a single floating-point value, use memory copies. Because the cost of memory copy for small variables is insignificant, we assume for the purposes of memory analysis that the assignment of one variable to another is always a pointer assignment. The *copy correction* decrements the identifier counts to properly address pointer assignment.

The corrections discussed so far involve decrementing counts to properly account for particular identifier occurrences. A different form of correction is required in `r = A*w;`. Here SLAMM needs additional information about the types of entity represented by `A` and `w`. For example, `A` may be a sparse or dense matrix. Type information is only available to the MATLAB interpreter. We address our lack of type information by transforming certain operators into function calls. For example, to apply the *special operator correction* we transform the expression `r=A*w` to the equivalent `r=mtimes(A,w)`. A profiled version of `mtimes` calculates the data movement based on the determination of type at runtime.

2.3 Translating SLAMM to MATLAB

Figure 3 illustrates the translation of Figure 2 by the SLAMM processor. The oval boxes in Figure 3 represent additional code blocks introduced to carry out the dynamic analysis. Note that the SLAMM directives, being MATLAB comments, can be copied into the output as documentation.

`Header` contains the definitions and initialization of MATLAB structures used to accumulate the information for each scope and identifier.

A `SizeOf` code block is inserted just after an assignment is made to a variable. It typically contains a single call to the MATLAB `whos` function. Here is the assignment and `SizeOf` block in scope B2 of Figure 3:

```
new = [r,b,3];
[slm_new] = whos('new');
```

This captures the type, extent, and required memory storage for `new` in a generated variable (`slm_new`) named for the user's variable. Field `bytes` of the structure created by the `whos('new')` call specifies the amount of storage occupied by variable `new`.

The purpose of an exclusive memory analysis block is to accumulate information from a particular scope. Each range requires an exclusive memory analysis code block. The exclusive memory analysis block for B1 (shown in Figure 3) contains (among other things):

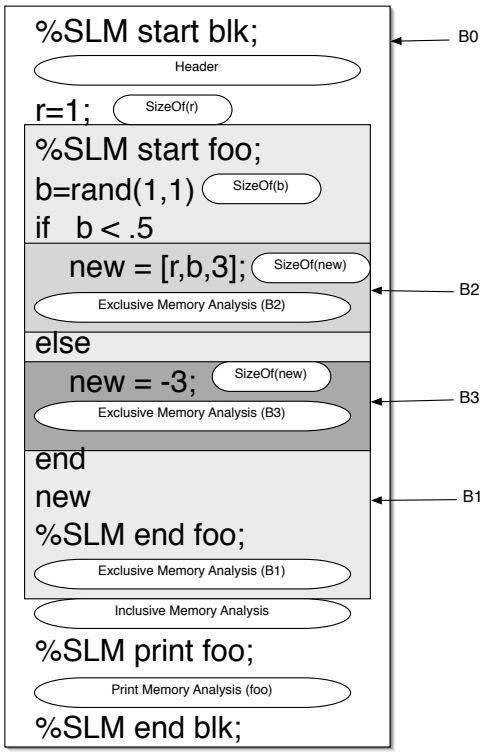


Fig. 3. Translation of code in Figure 2

```
slm_foo.wsl = slm_foo.wsl + slm_new.bytes + slm_b.bytes;
slm_foo.wss = slm_foo.wss + slm_b.bytes;
```

This calculates the amount of data loaded from the memory hierarchy by the `if`-expression and the fetch of `new`, accumulating the result in field `wsl` of the structure `slm_foo`. Similarly, the amount of data stored to the memory hierarchy (in the assignment to `b` at the beginning of the block) is recorded in field `wss`. Each of these assignments has two components: the total amount loaded or stored before this execution of the code in B1 (e.g., `slm_foo.wss`) and the amount loaded or stored during this execution (e.g., `slm_new.bytes`). The former is initialized to zero in `Header`, the latter is a constant determined by `whos`.

Note that this calculation accounts only for the accesses in the scope of the bindings of B1. Both `new` and `b` have defining occurrences in B2, and thus B2 constitutes a hole in the scope of the B1 bindings for those identifiers. Memory access information associated with the assignment in B2 is accumulated by code in the exclusive memory analysis code block B2.

In general, each term defining a variable access is multiplied by the count

of occurrences described in Section 2.2. This count is 1 for all variable accesses in Figure 3, and therefore SLAMM omits it.

The purpose of an inclusive memory analysis block is to gather the data from all of the exclusive memory analysis blocks in a function or program. It is generated at the end of the text of that function or program, where it will be executed exactly once. The inclusive memory analysis block of Figure 3 contains (among other things):

```
slm_foo.wsl = slm_foo.wsl + slm_elseLn7.wsl;
slm_foo.wsl = slm_foo.wsl + slm_ifLn5.wsl;
```

Here `ifLn5` and `elseLn7` are SLAMM's names for B2 and B3, respectively.

`Print Memory Analysis(foo)` consists of a call to a library function called `SlmPrtAnalysis` with an appropriate structure as its argument.

SLAMM differentiates between identifiers that correspond to functions and identifiers that correspond to variables. Functions are further classified into those that provide a significant contribution to data movement and those that do not. We refer to those functions that provide a significant contribution to data movement as *profiled functions*. SLAMM maintains the proper classification in a manually constructed table for a collection of commonly used built-in MATLAB functions. For example, the MATLAB function `size`, which determines the extent of a variable, typically does not contribute to data movement but the function `qr`, which calculates the QR factorization of an input matrix, certainly does.

Two types of code transformations are necessary for profiled functions. The first involves changing the function call, while the second involves changes to the function itself.

SLAMM transforms a function call to a profiled built-in MATLAB function by prefixing the string `SLM` to its name and adding an output argument. The additional output argument is a MATLAB structure containing the SLAMM-calculated memory analysis. The profiled function's contribution to data movement is subsequently accumulated in the appropriate exclusive memory analysis code block of the caller.

For profiled functions that return multiple output arguments, the code transformation only requires the addition of one extra output argument. For profiled functions that return a single output argument, additional code transformation may be necessary. For example, a single expression that uses the output argument of a profiled function as an operand requires the generation of multiple sub-expressions. All sub-expressions are linked by temporary variables. The generated MATLAB for the expression $t = sin(r) + cos(z)$, where $r, z, t \in \mathbb{R}^n$ is:

```
[slm_L1C5, slm_sin__L1C5] = SLMsin(r);
[slm_L1C14,slm_cos__L1C14] = SLMcos(z);
t = slm_L1C5 + slm_L1C14,
```

Here the single original expression $t = sin(r) + cos(z)$ is broken into three separate statements. The temporary variables `slm_L1C5` and `slm_L1C14` are the original output arguments of the *sin* and *cos* functions respectively and are used to calculate the expected result `t`. The structures `slm_sin__L1C5` and `slm_cos__L1C14` contain the results of the memory analysis of the *sin* and *cos* functions respectively.

The function call transformation requires the generation of new versions of the profiled functions. SLAMM provides a collection of profiled functions for all necessary built-in MATLAB functions. Each consists of a call to the original function and the assignment of the memory analysis structures. For a user-supplied MATLAB function, the SLAMM language processor generates the various memory analysis code blocks described above and alters the name and output arguments appropriately.

3 An Application of SLAMM

Automated memory analysis provides both the ability to evaluate the memory efficiency of a particular design choice rapidly during the design phase and the ability to improve the memory efficiency of a pre-existing solver. We illustrate both applications by using SLAMM to reduce the execution time of the Parallel Ocean Program (POP) [32], a global ocean model developed at Los Alamos National Laboratory. POP, which is used extensively as the ocean component of the Community Climate System Model [8], uses a preconditioned conjugate gradient solver to update surface pressure in the barotropic component. Parallelism on distributed memory computers is supported through the Message Passing Interface (MPI) [34] standard.

We used POP version 2.0.1 [29] to examine data movement in the preconditioned conjugate gradient solver using the `test` and `gx1v3` grids. The `test` grid is a coarse grid provided with the source code to facilitate the porting of POP to other compute platforms. POP with the `gx1v3` grid, which has one degree separation between grid points at the equator, is higher resolution than the `test` grid and represents 20% of the total compute cycles at the National Center for Atmospheric Research (NCAR) [4].

POP uses a three-dimensional computational mesh. The horizontal dimensions are decomposed into logically rectangular two-dimensional (2D) blocks [21]. The computational mesh is distributed across multiple processors by placing one or more 2D blocks on each processor. The primary advantage of the 2D data structure is that it provides a regular stride-one access for the matrix-vector multiply. The disadvantage of the 2D data structure within the conjugate gradient solver is that it includes a potentially large number of grid points that represent land. In effect, a number of explicitly stored zeros are added to the matrix. An alternative 1D data structure that uses compressed sparse row storage would avoid the inclusion of land points but would introduce indirect addressing. Additional details concerning the changes in data

Table 2
The data movement for a single iteration of preconditioned conjugate gradient solver in POP using the `test` grid. The values of WSL_P and WSL_M are in Kbytes

Solver Implementation	WSL_P	version	Ultra II WSL_M	Ultra II error	POWER4 WSL_M	POWER4 error	R14K WSL_M	R14K error
PCG2+2D	4902	v1	5163	5.3%	5068	3.4%	5728	16.9%
		v2	4905	0.1%	4865	-0.7%	4854	-1.0%
PCG2+1D	3218		3164	-1.7%	3335	3.7%	3473	7.9%
Reduction	34%		39%		34%		39%	

structures are provided in [11].

We used SLAMM to evaluate the required data movement for conjugate gradient solvers based on 1D and 2D data structures. We describe two types of data movement, the predicted data movement (WSL_P) and the measured data movement (WSL_M). WSL_P is predicted by SLAMM, and WSL_M is measured using hardware performance counters and refers to data loaded from the L2 to the L1 cache. For the 1D data structure, we used the PCG solver routine provided by MATLAB. We wrote a new PCG solver in MATLAB which used the same 2D data structures as in POP. Using the SLAMM directives described in Section 2, we created a region for one iteration of each of the algorithms. WSL_P for the algorithm with the 2D data structure (PCG2+2D) and the 1D data structure version (PCG2+1D) for the `test` grid are provided in the second column of Table 2. We tested both a naive implementation of PCG2+2D (v1) and an optimized one (v2) described later in this section. The SLAMM prediction is the same for both.

SLAMM predicts that the use of the 1D data structure reduces data movement by 34% versus the existing 2D data structure. Based on the expectation of a 34% reduction in data movement, we implemented the 1D data structure in POP. If SLAMM had predicted a minor reduction in data movement, or even an increase in data movement, the 1D data structure would have never been implemented. *A priori* analysis like this provides confidence that the programming time to implement code modifications or new algorithms is not wasted.

To evaluate the quality of our implementations and to check the accuracy of the SLAMM-based predictions, we instrumented all versions of the solver with a locally developed performance profiling library (Htrace), which is based on the PAPI [27] hardware performance counter API. Htrace calculates data movement by tracking the number of cache lines moved through the different components of the memory hierarchy. We focus on three primary microprocessor compute platforms that provide counters for cache lines loaded from the memory hierarchy to the L1 cache: Sun Ultra II [24], IBM POWER 4 [3], and

Table 3
Description of the microprocessor compute platforms and their cache configurations

CPU	Ultra II	POWER4	R14K
Company	SUN	IBM	SGI
Mhz	400	1300	500
L1 Data-cache	32KB	32KB	32KB
L2 cache	4 MB	1440 KB	8 MB
L3 cache	–	32 MB	–

MIPS R14K [36]. A description of the cache configuration for each compute platform is provided in Table 3.

The measured data movement (WSL_M) is also presented in Table 2 for the 2D data structure implementations (PCG2+2D v1 and v2) and the 1D version (PCG2+1D) on each of the compute platforms. We report average data movement across ten iterations. While the discrepancies between the measured and predicted WSL for the PCS2+2D v1 solver are minimal for both the Ultra II and POWER4 platforms, the measured value of 5728 Kbytes for the R14K is 17% greater than the predicted value of 4902 Kbytes. The difference in data movement between the three compute platforms may be due to additional code transformations performed by the Ultra II and POWER4 compilers. That the PCG2+2D v1 solver is loading 17% more data from the memory hierarchy than necessary on the R14K is an indication that it is possible to improve the quality of the implementation.

While SLAMM provides an indication of a potential performance problem, it does not isolate the source of the problem. Locating and addressing the problem is still the responsibility of the software developer. An examination of the source code for the PCG2+2D v1 solver indicates that a minor change to the dot product calculation reduces data movement. Code blocks for version v1 and v2 of the solver are provided in Figure 4. The function `operator` applies the local matrix product, and the array `LMASK` is an array that masks out points that correspond to land points. In version v1 of the `do` loop, a temporary array `WORK0` is created that contains the point-wise product of two vectors `Q` and `P`. Outside the `do` loop, the dot product of `LMASK` and the `WORK0` array is calculated by `global_sum`. If the size of data accessed in the `do` loop is larger than the L1 cache, then a piece of the `WORK0` array at the end of the `do` loop is no longer located in the L1 cache and must be reloaded to complete the calculation.

In the optimized version v2 of the `do` loop, we employ a scalar temporary `delta_local` to accumulate each block's contribution to the dot product of `Q` and `P`. We then use a function `local_sum` that applies the land mask to complete the dot product. Finally, we replace the function `global_sum` with

```
! ================================
! code block: solver v1
! ================================
do iblock=1,nblocks
     P(:,:,iblock) = Z(:,:,iblock) + P(:,:,iblock)*beta
     Q = operator(P,iblock)
     WORK0(:,:,iblock) = Q(:,:,iblock)*P(:,:,iblock)
enddo
delta=global_sum(WORK0,LMASK)

! ================================
! code block: solver v2
! ================================
delta_local=0.d0
do iblock=1,nblocks
     P(:,:,iblock) = Z(:,:,iblock) + P(:,:,iblock)*beta
     Q = operator(P,iblock)
     WORK0 = Q(:,:,iblock)*P(:,:,iblock)
     delta_local = delta_local + local_sum(WORK0,LMASK(:,:,iblock))
enddo
delta=gsum(delta_local)
```

Fig. 4. A code block that implements a piece of the PCG algorithm for versions v1 and v2.

a call to `gsum`. The subroutine `gsum`, when executed on a single processor, is an assignment of `delta_local` to `delta`. Because version v2 of the code block does not access `WORK0` outside the `do` loop, it potentially reduces data movement.

The WSL_M values in Table 2 for the v1 and v2 versions of the PCG2+2D solver on the R14K indicate that the rearranged dot product calculations reduce data movement by 18%. Note, that data movement is also reduced on the Ultra II and POWER4 platforms but to a lesser extent.

Table 2 also provides the relative error between predicted and measured data movement for the solvers. SLAMM predicts data movement to within an average error of 0.6% for the PCG2+2D v2 solver and within 4.4% for the PCG2+1D solver. (The deviation for the poorer quality PCG2+2D v1 is greater.) Table 2 indicates that the actual percentage reductions in data movement are very similar to the predicted reductions. The results in Table 2 clearly demonstrate that it is possible for automated memory analysis to predict the amount of data movement accurately and so to provide *a priori* knowledge of the memory efficiency of a particular design choice before implementation in a compiled language.

Note that it is entirely possible to perform the same analysis manually without SLAMM. While the calculation for the 2D case would be straightforward, the 1D case would be more complicated. In particular, an accurate

prediction of the amount of data movement for the 1D data structure would require detailed information about location of land points.

It is possible to estimate the difficulty of manual memory analysis by comparing the lines of code associated with the MATLAB prototype to the memory analysis code. Table 4 shows the number of non-comment lines for the 1D PCG prototype in MATLAB. For the 1D PCG prototype, the prototype grew from 41 lines to a total of 48 lines with the addition of seven %SLM directives. The SLAMM generated output code (prototype and memory analysis code) is 290 lines. In this case, memory analysis requires approximately seven times the number of code lines required by the algorithm alone. This illustrates that manual calculations are generally onerous and error prone and thus may not be performed regularly (or at all).

We next examine the impact of reducing data movement on execution time. The timestep of POP includes a baroclinic and a barotropic component. The barotropic component is composed of a single linear solver for surface pressure. We executed POP using the `test` grid on a single processor of each compute platform for a total of 20 timesteps. This configuration requires an average of 69 iterations of the PCG2 algorithm per timestep. Table 5, gives the barotropic execution time in seconds using the three implementations of the solver. We include the execution time for the initial implementation of the solver using 2D data structures to accurately reflect the overall impact automated memory analysis has on execution time. Note that the PCG1+1D solver consistently has a lower execution time than either of the 2D data structure based solvers. The last row in Table 5 contains the percentage reduction in barotropic execution time for the PCG2+2D v1 versus the PCG2+1D solver. Table 5 indicates that code modifications either evaluated or identified by automated memory analysis reduce execution time by an average of 46%. Curiously, a comparison of Tables 2 and 5 indicates that the percentage reduction in execution time is even larger than the percentage reduction in data movement. The reason for the discrepancy is unclear.

We next examine the impact of the 1D data structures on parallel execution time on POWER4 platform using the `gx1v3` grid. We focus on the execution time of POP on 64 POWER4 processors. This is a typical configuration of POP that consumes approximately 2.4 million CPU hours every year at NCAR. We wrote a generalized gather-scatter routine to provide par-

Table 4
Source code lines for MATLAB prototype.

Developer generated Source Lines		SLAMM generated prototype and analysis code
original	%SLM	
41	7	290

Table 5
Barotropic execution time for 20 timesteps of POP in seconds using the `test` grid on a single processor

Solver implementation	Ultra II	POWER4	R14K
PCG2+2D v1	21.17	4.57	8.58
PCG2+2D v2	20.49	4.01	7.97
PCG2+1D	12.74	2.11	4.61
Reduction	39%	54%	46%

allel execution under MPI for the 1D version of the solver. Recently, an alternative preconditioned conjugate gradient algorithm that uses a single dot product [9] (PCG1) was added to POP. The PCG1 algorithm provides a performance advantage for parallel execution because it eliminates one of the distributed dot product calculations. We configured POP to execute a total of 200 timesteps on 64 IBM POWER4 processors, with an average of 151 iterations per timestep. Note that, while the cost of the solver is important, it does does not dominate the total cost of a timestep. The execution time for the time stepping loop in seconds for the four solver implementations is provided in Table 6. These results indicate that use of the PCG1+1D solver versus the PCG1+2D solver reduces total POP execution time by 9%. A 9% reduction in total execution time of POP is significant because that model has already been extensively studied and optimized [21,33]; the additional 9% translates into a reduction of 216,000 CPU hours per year at NCAR.

4 Conclusion

Data movement is an important characteristic of a numerical algorithm running on today's computers, contributing significantly to that algorithm's running time. The effects of design decisions on data movement should therefore be explored before undertaking a costly implementation. Predictions of data movement are possible by adding instrumentation to a MATLAB prototype of the algorithm. As in the case of the partial derivative calculation problem described by Gabriel nearly forty years ago, that process is at best a nuisance

Table 6
Total execution time for 200 timesteps with `gx1v3` grid on 64 POWER4 processors.

	Solver Implementation			
	PCG2+2D v1	PCG1+2D v1	PCG2+1D	PCG1+1D
total time (sec)	86.2	81.5	78.8	73.9

and at worst a common source of errors.

SLAMM automates the process of modifying the MATLAB prototype, reducing the cost of a test from days to minutes. It encapsulates knowledge about the characteristics of variable access and about strategies for accumulating information during a prototype run. If more detailed knowledge and better strategies evolve over time, they can be incorporated into SLAMM without the need for user re-training.

Most of the difficulty in automating the instrumentation process lies in the common tasks of analyzing the text of the prototype to select the appropriate instrumentation code. Because of advances in compiler technology, those analysis tasks can be described by declarative specifications from which code can be produced automatically. Doing so dramatically reduces the cost of developing a tool like SLAMM.

We have shown that SLAMM can be used to guide the performance improvement of large codes that are in current use as well as in the design of new algorithms. It provides one more example of the utility of source-to-source translators taking over the tedious and error-prone aspects of software development.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. J. Dongarra, J. DuCroz, A. Greenbaum, and S. Hammarling. *LAPACK Users' Guide*. SIAM, 1992.

[2] A. Baker, J. Dennis, and E. R. Jessup. Toward memory-efficient linear solvers. In J.M.L.M. Palma, J. Dongarra, V. Hernandez, and A. A. Sousa, editors, *VECPAR '2002, Fifth International Conference on High Performance Computing for Computational Science: Selected Papers and Invited Talks*, volume 2565 of *Lecture Notes in Computer Science*, pages 315–327. Springer, Berlin, 2003.

[3] S. Behling, R. Bell, P. Farrell, H. Holthoff, F. O'Connell, and W. Weir. *The POWER4 Processor Introduction and Tuning Guide*. IBM Redbooks, November 2001. http://www.redbooks.ibm.com.

[4] T. Bettge. National Center for Atmospheric Research, Dec. 2005. Personal Communication.

[5] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *ICS '97: Proceedings of the 11th International Conference on Supercomputing*, pages 340–347, New York, NY, USA, 1997. ACM Press.

[6] L. Carter, J. Ferrante, and S. F. Hummel. Hierarchical tiling for improved superscalar performance. In *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*, pages 239–245, Washington, DC, USA, 1995. IEEE Computer Society.

[7] A. T. Chronopoulos and C. W. Gear. S-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25:153–168, 1989.

[8] W. D. Collins, C. M. Bitz, M. L. Blackmon, G. B. Bonan, C. S. Bretherton, J. A. Carton, P. Chang, S. C. Doney, J. J. Hack, T. B. Henderson, J. T. Kiehl, W. G. Large, D. S. McKenna, B. D. Santer, and R. D. Smith. The community climate system model: CCSM3. *Journal of Climate: CCSM Special Issue*, 11(6), 2005.

[9] E. F. D'Azevedo, V. L. Eijkhout, and C. H. Romaine. Conjugate gradient algorithms with reduced synchronization overhead on distributed memory multiprocessors. Technical Report 56, Lapack Working Note, August 2002.

[10] J. M. Dennis. *Automated Memory Analysis: Improving the Design and Implementation of Iterative Algorithms*. PhD thesis, University of Colorado, Boulder, July 2005.

[11] J. M. Dennis and E. R. Jessup. Applying automated memory analysis to improve iterative algorithms. *SIAM Journal of Scientific Computing*, 29(5):2210–2223, 2007.

[12] J. Dongarra, J. DuCroz, S. Hammarling, and I. Duff. Algorithm 679: A set of level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 16:18–28, 1990.

[13] Eli: An integrated toolset for compiler construction. Documentation, examples, download from http://eli-project.sourceforge.net/.

[14] Engineering and Scientific Subroutine Library (ESSL) and Parallel ESSL. http://www.ibm.com/systems/p/software/essl.html, 2007.

[15] M. Gabriel. Special-purpose language for least-squares fits. Technical Report ANL-7495, Applied Mathematics Division, Argonne National Laboratory, September 1968.

[16] M. Gabriel. A symbolic derivative taker for a special-purpose language for least-squares fits. Technical Report ANL-7628, Applied Mathematics Division, Argonne National Laboratory, February 1970.

[17] K. Goto and R. van de Geijn. High-performance implementation of the level-3 BLAS. Technical Report TR-2006-23, The University of Texas at Austin, Department of Computer Sciences, 2006.

[18] R. W. Gray, S. P. Levi, V. P. Heuring, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–130, 1992. http://eli-project.sourceforge.net/.

[19] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Toward realistic performance bounds for implicit CFD codes. In A. Ecer et al., editor, *Proceedings of Parallel CFD'99*, pages 233–240. Elsevier, 1999.

[20] Intel Math Kernel Library. http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm, 2007.

[21] P. W. Jones, P. H. Worley, Y. Yoshida, J. B. III White, and J. Levesque. Practical performance portability in the Parallel Ocean Program (POP). *Concurrency Comput. Prac. Exper*, 17:1317–1327, 2005.

[22] U. Kastens, A. M. Sloane, and W. M. Waite. *Generating Software from Specifications*. Jones and Bartlett, Sudbury, MA, 2007.

[23] U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.

[24] Sun Microsystems. The Ultra2 architecture: Technical white paper, 2005. http://pennsun.essc.psu.edu/customerweb/WhitePapers/.

[25] A. A. Mirin, R. H. Cohen, B. C. Curtis, W. P. Dannevik, A. M. Dimits, M. A. Duchaineau, D. E. Eliason, D. R. Schikore, S. E. Anderson, D. H. Porter, P. R. Woodward, L. J. Shieh, and S. W. White. Very high resolution simulation of compressible turbulence on the ibm-sp system. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 70, New York, NY, USA, 1999. ACM Press.

[26] D. P. O'Leary. The block conjugate gradient algorithm and related methods. *Linear Algebra and its Applications*, 29:293–322, 1980.

[27] PAPI: Performance Application Programming Interface: User's Guide. http://icl.cs.utk.edu/papi, 2005.

[28] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yellick. A case for intelligent RAM. *IEEE Micro*, pages 34–44, March/April 1997.

[29] The Parallel Ocean Program (POP). http://climate.lanl.gov/Models/POP, 2006.

[30] Y. Saad. Krylov subspace methods for solving large unsymmetric linear systems. *Mathematics of Computation*, 37(155):105–126, 1981.

[31] H. D. Simon. *The Lanczos algorithm for solving symmetric linear systems*. PhD thesis, University of California, Berkeley, April 1982.

[32] R. D. Smith, J. K. Dukowicz, and R. C. Malone. Parallel ocean general circulation modeling. *Physica D*, 60:38–61, 1992.

[33] A. Snavely, X. Gao, C. Lee, L. Carrington, N. Wolter, J. Labarta, J. Gimenez, and P. Jones. Performance modeling of HPC applications. In *Parallel Computing (ParCo2003)*, 2003.

[34] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference: Volume 1, The MPI Core*. The MIT Press, 2000.

[35] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proceedings of Supercomputing 1992*, 1992.

[36] S. Vellas. Scalar Code Optimization I, 2005.
http://sc.tamu.edu/help/origins/sgi_scalar_r14k_opt.pdf.

[37] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

Interfacing Concepts:
Why Declaration Style Shouldn't Matter

Anya Helene Bagge [1] Magne Haveraaen [2]

Department of Informatics, University of Bergen, Norway

Abstract

A *concept* (or *signature*) describes the interface of a set of abstract types by listing the operations that should be supported for those types. When implementing a generic operation, such as sorting, we may then specify requirements such as "elements must be comparable" by requiring that the element type models the Comparable concept. We may also use axioms to describe behaviour that should be common to all models of a concept.

However, the operations specified by the concept are not always the ones that are best suited for the implementation. For example, numbers and matrices may both be addable, but adding two numbers is conveniently done by using a return value, whereas adding a sparse and a dense matrix is probably best achieved by modifying the dense matrix. In both cases, though, we may want to pretend we're using a simple function with a return value, as this most closely matches the notation we know from mathematics.

This paper presents two simple concepts to break the notational tie between implementation and use of an operation: *functionalisation*, which derives a canonical pure functional version from a procedure; and *mutification*, which translates calls using the functionalised declarations into calls to the implemented procedure.

Key words: concepts, functions, procedures, mutification, axioms, imperative vs functional, concept-based programming

1 Introduction

Concepts is a useful feature for generic programming, allowing programmers to specify the interface and behaviour of abstract data types. The concept feature was introduced to C++0x (the upcoming C++ standard revision) in order to give clearer error messages in templated code, and to provide a way to more easily glue together generic code.

[1] `http://www.ii.uib.no/~anya/`
[2] `http://www.ii.uib.no/~magne/`

In this paper we will look at concepts in the context of the Magnolia Programming Language, an experimental language loosely based on C++. Concepts form an integral part of Magnolia, and the programmer is encouraged to specify code dependencies in terms of concepts, and use concepts to specify the interface of implementation modules. Concepts are used to hide implementation details, so that one module may be replaced by another modelling the same concept. Even 'standard' data types like numbers and strings are handled through concepts, allowing for the possibility of replacing or using different implementations of basic data types.

If we are to hide away different implementations behind a common interface, we must consider that different situations call for different programming styles. Operations on primitive data types, for example, are conveniently handled using return values, whereas the corresponding operation on a big data structure may be better handled by updating it.

Also, the needs of a library implementation may differ from the needs of the code that uses it. For example, a mathematical problem may be easily expressed with function and operator calls, whereas a communication protocol could be better expressed as a sequence of procedure calls. As a library user [3], however, you are locked to the style the library writer chooses to support—and your preferred style may be less convenient to implement, or less efficient, and thus not supported.

C++ is an example of a language where the proliferation of notational variants is especially bad (or good, depending on your point of view) – with both member and non-member functions, operator overloading, and a multitude of parameter passing modes (value, reference, const reference, pointers). Functional languages like ML and Haskell are less problematic since the languages are already restricted to functional-style declaration forms.

Generic and generative techniques are hampered by a proliferation of declaration forms (prototypes, in C++ terminology) for implementations of the same abstract idea. For instance, implementing generic algorithms, such as a generic `map` operation for arrays, is made more difficult when the declaration forms of the element operations vary—so we end up with a multitude of different `map` implementations, e.g., one for arrays of numbers and one arrays of matrices.

For C++0x, the problem of having many different declaration forms for what is essentially the same operation is solved by allowing the implementer to add explicit glue code in the *concept map* (a declaration that some given types model a given concept). In constrained template code, the user will use the declarations from the concept and need not know anything about how the implementation is declared. Note, however that this convenience is available in constrained template code only.

[3] *User* refers to the programmer who is using a library, and the word *implementer* to the programmer who has implemented it. The *end user* who runs the finished software product is outside the scope of our discussion.

In this paper, we show how this is solved in Magnolia by separating the *implementation signature* from the *use signature*, allowing different call styles to be used independently of the implementation style. By letting the compiler translate between different call styles, we gain increased flexibility for program processors—of both the human and software variety. We call these translations *functionalisation* and *mutification*.

We will start out by giving an intuition about our method, before we define it more formally later in the paper. We illustrate our points with examples in Magnolia, which has functionalisation and mutification as basic, built-in features.

Magnolia supports imperative *procedures*, which are allowed to update their arguments, and pure *functions*, which are not. A sample Magnolia procedure declaration looks like this:

```
procedure fib(upd int n);
```

The procedure `fib` takes a single parameter, an ***upd****atable integer*, which is used both for input and for returning a result.

Functionalisation takes a procedure declaration, and turns it into a function declaration. The functionalisation of `fib` is:

```
function int fib(int n);
```

—a function with an integer parameter, returning an integer. This is the declaration that would typically be used in a concept. A different declaration of the `fib` procedure—like this, for example,

```
procedure fib(obs int n, out int r);
```

—which ***obs****erves* the parameter `n` and ***out****puts* the parameter `r`, yields the same functionalisation.

Functionalising the procedure declaration gives us a function we can call from Magnolia expressions, but no implementation of that function. This is where *mutification* comes into the picture. Mutification takes an expression using a functionalised procedure, and turns it into a statement that calls the procedure directly. Here we show a call for each of the declarations above, the functional form (in the middle) can be transformed to either of the procedural calls shown.

```
y = x;
call fib(y);        ←  y = fib(x);        →  call fib(x,y);
```

Why would we want to implement an algorithm with one declaration form and use it with another? As explained above, the implementation side is often dictated by what fits with the problem or algorithm—in-place sorting and updating matrix operations, for instance, will require less memory and may run faster than versions that construct new objects. There are several reasons why an algebraic style is useful on the user side:

Flexibility: Multiple implementations with different characteristics can be hidden behind the same interface. This is particularly useful in generic code and when building a basis of interchangeable components.

Ease of reasoning: Not just for humans, but also for compilers and tools. For example, axiom-based rewrite rules [2,8,11] allow programmers to aid the compiler's optimisation by embedding simplification rules within a program. Such rules are much harder to express with statements and updating procedures.

Notational clarity: certain problems are most clearly expressed in an algebraic or functional style, which is close to the notational style of mathematics, a notation developed over the centuries for clarity of expressions. Many formal specification languages, e.g., the Larch Shared Language [6] and CASL [9] use the functional style due to its clarity. A program written in an algebraic style is easy to relate to a formal specification.

Using a single declaration form makes it possible to state rules such as "+ and × are commutative for algebraic rings", and have it apply to all types satisfying the properties of algebraic rings (i.e., all types modelling the concept *Ring*) – independently of whether the implementation uses return values or argument updates.

We may in principle choose any declaration form as the common, canonical style—we have chosen the algebraic style for the reasons above, and because it is less influenced by implementation considerations (such as the choice of which argument(s) should output the result). As we shall see, this choice does not mean that we must enforce an algebraic programming style.

We will now proceed with a deeper discussion of the matter. The rest of this paper is organised as follows: First, we introduce the necessary features of the Magnolia language (Sect. 2). Then we define mutification and functionalisation, and explain how they are applied to Magnolia programs (Sect. 3). We continue by discussing some limitations and pragmatic considerations, and the benefits and possibilities of the approach (Sect. 4).

2 The Magnolia Language

This section gives a brief overview of the Magnolia Programming Language. Magnolia is based on C++, with some features removed, some features added, and some changes to the syntax. We have designed it to be easier to process and write tools for than C++ (this is actually our main reason for using a new language, rather than working with C++), while being similar enough that we can easily compile to C++ code and use high-performance C++ compilers.

The following features are the ones that are relevant for this paper:

Procedures have explicit control over their inputs and outputs: they are allowed to modify their parameters, according to the *parameter modes* given in the procedure declaration. The available modes are `obs`*erve* for input-only arguments, `upd`*ate* for arguments that can be both read and written to, and `out`*put* for output-only arguments. These modes describe the data-flow characteristic of the procedure – which values the result may depend on, and

which variables may be changed by the procedure. The parameter passing mode (e.g., by value, by reference, by copying in/out) is left undefined, so the compiler is free to use the most efficient passing mode for a given data type. Procedures have no return values, the result is given by writing to one or more arguments. Procedure calls use the `call` keyword, so they are easy to distinguish from function calls.

Functions have a single return value, which depends solely on the arguments. They are not allowed to modify arguments. Operators can be overloaded, and are just fancy syntax for function calls.

Functions and procedures are known collectively as *operations*.

Classes are similar to C++ structs and classes, though there is no difference between member and non-member operations – everything is treated as non-members. There is no dynamic dispatch or inheritance yet, and we won't consider that in this paper.

Concepts describe the interfaces of types by listing some required operations, axioms on the operations and possibly other requirements. A set of types is said to model a concept if the operations are defined for those types and the requirements are satisfied. For example, the following defines a simple 'Indexable' concept:

```
concept Indexable<A,I,E> {
  E getElt(A, I);
  A setElt(A, I, E);

  axiom getset(A a, I i, E e) {
    assert getElt(setElt(a, i, e), i) == e;
  }
```

Indexable has three types, and array-like type `A`, an index type `I` and an element type `E`. The concept defines two functions, `getElt` and `setElt`[4], and a simple axiom relating the functions.

Templates: Generic programming is done through a template facility similar to C++'s. Template parameters may be constrained using concepts, so that only types modelling a given concept are acceptable as template arguments.

No Aliasing: Aliasing makes it difficult to reason about code, because it destroys the basic assumption that assigning to or updating one variable will not change the value of another. Functional languages avoid this problem by simply banning the whole idea of modifying variables. We feel disallowing modification is too high a price to pay, particularly when working with numerical software and large data structures, so Magnolia has a set of rules designed to prevent aliasing while still having most of the freedom of imperative-style code:

- No pointers or references. Any data structures that need such features

[4] We have avoided using operators to keep the syntax simple

must be hidden behind a 'clean' interface. The programmer must take responsibility for ensuring that the implementation is safe.
- There is no way to refer to a part of a data structure. For example, you can't pass an element of an array as a procedure parameter – you must either pass the entire array, or the *value* of the element. Thus, changing an object field or an array element is an operation on the object or array itself (unlike in C++, where fields and elements are typically l-values and can be assigned to directly). This is why the `setElt` operation in the Indexable concept above is declared as a function returning an array, and not returning a reference to an element as is typical in C++.
- If a variable is passed as an `upd` or `out` argument to a procedure, that variable cannot be used in any other argument position in the same call.

The former two rules ensure that no aliasing occurs within an operation, and the latter rule ensures that aliasing does not occur when a call is made.

3 Relating Functions and Procedures

We will now establish a relationship between Magnolia functions and procedures, so that each procedure declaration has corresponding function declarations given by functionalisation (Def. 3.1), and every expression has a corresponding sequence of procedure calls given by mutification (Def. 3.2).

3.1 Functionalisation of Declarations

Definition 3.1 *Functionalisation*, F, maps a procedure declaration to function declarations. This makes procedures accessible from expressions, at the signature level. Since a procedure can have multiple output parameters, and a function can only have one return value, we get one function for each output parameter of the procedure (numbered 1 to i):

$$\mathrm{F_i}(\mathsf{Proc}(n, \boldsymbol{q})) = \mathsf{Fun}(n_i, \mathrm{Out}(\boldsymbol{q})_i, \mathrm{In}(\boldsymbol{q})) \tag{1}$$

For clarity, we use abstract syntax in the definitions, with Proc (name, proc-parameter-list) being a procedure declaration, and Fun (name, return-type, fun-parameter-list) being a function declaration. In and Out gives the input and output parameters of a procedure, respectively:

$$\mathrm{In}(\boldsymbol{q}) = [t \mid \langle m, t\rangle \leftarrow \boldsymbol{q}, m \in \{\texttt{obs}, \texttt{upd}\}]$$
$$\mathrm{Out}(\boldsymbol{q}) = [t \mid \langle m, t\rangle \leftarrow \boldsymbol{q}, m \in \{\texttt{out}, \texttt{upd}\}]$$

where m is the parameter mode and t is the parameter type.

We can then obtain the list of functions corresponding to a procedure[5]:

$$\mathrm{F}(\mathsf{Proc}(n, \boldsymbol{q})) = [\mathrm{F}_i(\mathsf{Proc}(n, \boldsymbol{q})) \mid i = 1 \ldots \mathrm{len}(\mathrm{Out}(\boldsymbol{q}))] \tag{2}$$

[5] Using list comprehension notation (similar to set notation), as in Haskell or Python.

Note the similarity between functionalisation and standard techniques for describing the semantics of a procedure with multiple return values. This is the link between the semantics of the procedure and the functionalised version, and the key in maintaining semantic correctness between the two programming notations.

For example, the following procedures:

```
procedure plus(upd dense x, obs sparse y);
procedure plus(obs int x, obs int y, out int z);
procedure copy(obs T x, out T y);
```

functionalise to the following functions:

```
function dense plus(dense x, sparse y);
function int plus(int x, int y);
function T copy(T x);
```

which is what would be used in a concept declaration. We keep the name space of functions and procedures, as well as their usage notations (expressions versus calls), distinct, thus avoiding overloading conflicts [6].

Note that the inverse operation—obtaining a procedure from a function—is not straight-forward, since there are many different mode combinations for the same function declaration. However, we could define a *canonical proceduralisation*, P, in which every function is mapped to a procedure with one `out` parameter for the return value and one `obs` parameter for each parameter of the function:

$$\mathrm{P}(\mathsf{Fun}(n, t', [t_1, \ldots, t_n]) = \mathsf{Proc}(n, [\langle \texttt{obs}, t_1\rangle, \ldots, \langle \texttt{obs}, t_n\rangle, \langle \texttt{out}, t'\rangle]) \qquad (3)$$

3.2 Mutification

Definition 3.2 [Mutification of Assignment] Mutification turns a sequence of function calls and assignments into a procedure call. Given a procedure $p = \mathsf{Proc}(n, \boldsymbol{q})$:

$$\mathcal{M}(\overline{\mathsf{Assign}(\boldsymbol{y}, \mathsf{Apply}(\boldsymbol{f}, \boldsymbol{x}))}) = \boldsymbol{i}\ ;\ \mathsf{Call}(\mathsf{Proc}(n, \boldsymbol{q}), \boldsymbol{x'})], \text{ where } \mathrm{F}(\mathsf{Proc}(n, \boldsymbol{q})) = \boldsymbol{f},$$
$$\text{and } \langle \boldsymbol{x'}, \boldsymbol{i}\rangle = \mathrm{unzip}\left(\boldsymbol{q} \begin{bmatrix} \langle \texttt{obs}, s\rangle \to \langle x, \mathsf{Nop}\rangle \text{ if } x \notin \boldsymbol{y} \\ \qquad \text{where } x = \downarrow \boldsymbol{x} \\ \langle \texttt{obs}, s\rangle \to \langle t, \mathsf{TmpVar}(t, x)\rangle \text{ if } x \in \boldsymbol{y} \\ \qquad \text{where } x = \downarrow \boldsymbol{x} \\ \langle \texttt{out}, s\rangle \to \langle \downarrow \boldsymbol{y}, \mathsf{Nop}\rangle \\ \langle \texttt{upd}, s\rangle \to \langle y, \mathsf{Assign}(y, \downarrow \boldsymbol{x})\rangle, \\ \qquad \text{where } y = \downarrow \boldsymbol{y} \end{bmatrix}\right)$$
$$\text{and } y_j \equiv y_k \iff j = k \text{ for all } y_j, y_k \in \boldsymbol{y} \qquad (4)$$

[6] We still need to give the different functions resulting from one procedure distinct names.

The pattern $\overline{\mathsf{Assign}(\boldsymbol{y}, \mathsf{Apply}(\boldsymbol{f}, \boldsymbol{x}))}$ recognises a sequence of assignments, one for each `upd` or `out` argument of p. The list of functions called, $\boldsymbol{f}$, must match the functionalisation of p, in sequence. Dummy assignments can be inserted to accomplish this, if no suitable instructions can be moved from elsewhere. All variables ($\boldsymbol{y}$) assigned to must be distinct.

We then construct a new argument list $\boldsymbol{x'}$ and a list of setup statements $\boldsymbol{i}$ by examining the formal parameter list of p, picking ($\downarrow$) an argument from $\boldsymbol{x}$ in case of `obs`, picking from $\boldsymbol{y}$ in case of `out`, and picking from $\boldsymbol{y}$ and generating an assignment $\mathsf{Assign}(y, x)$ in case of `upd`. Nop denotes an empty instruction, TmpVar creates a temporary variable, and Assign denotes an assignment. To avoid aliasing problems, a temporary is needed to store the value of an `obs` argument which is also used for output.

Generating assignments for `upd` arguments is necessary, since the variables $\boldsymbol{y}$ may have different values than the corresponding variables in the original argument list. This may generate redundant $\mathsf{Assign}(y, y)$ instructions when y is already in an update position—these can trivially be eliminated at a later stage.

For example, if we have a procedure $p(\texttt{out}\, t_1, \texttt{upd}\, t_2, \texttt{obs}\, t_3)$ and $f_1, f_2 = F(p)$:

```
a = f_1(3, 2);          →      b = 3;
b = f_2(3, 2);                 call p(a, b, 2);
```

3.3 Functionalisation of Procedure Calls

Now, we've seen how we can turn a function call and assignment into a procedure call. Can we do the inverse operation? Certainly, but there are a few caveats.

With mutification, we can rely on functionalisation of procedure declarations to obtain function declarations. But since there are plenty of ways to proceduralise a function declaration, we will have to declare which declaration(s) we want—or, at the very least, settle for some sort of canonical procedure declaration, as sketched in Sect. 3.1.

The second snag is that we're giving up some control over the program—for example the creation and deletion of intermediate variables.

However, since we argue that expression-based programs are much easier to analyse and process by tools, it should be useful to translate procedure calls to function calls to obtain something as close to a pure expression-based program as possible – even if we will eventually mutify it again to obtain imperative code. The choice of language form to work on becomes one of convenience. Our experience is mostly with high-level optimisations that take advantage of algebraic laws – which is a prefect fit for algebraic-style Magnolia. The same can be seen in modern optimising compilers, which will typically transform low-level code to and from Static Single Assignment (SSA) form depending on which optimisation is performed.

Definition 3.3 [Functionalisation of Procedure Calls] For a procedure $p = \mathsf{Proc}(n, \boldsymbol{q})$:

$$\mathcal{F}(\mathsf{Call}(p, \boldsymbol{x})) = [\mathcal{F}_i(\mathsf{Call}(p, \boldsymbol{x}))|i = 1...\,\mathrm{len}(\mathrm{Out}(\boldsymbol{q}))] \tag{5}$$
$$\mathcal{F}_i(\mathsf{Call}(p, \boldsymbol{x})) = \mathsf{Assign}(\boldsymbol{y}_i, \mathsf{Apply}(\mathrm{F}_i(p), \boldsymbol{x'})) \tag{6}$$

where $\boldsymbol{y} = \mathrm{Out}_{\boldsymbol{q}}(\boldsymbol{x})$ and $\boldsymbol{x'} = \mathrm{In}_{\boldsymbol{q}}(\boldsymbol{x})$. The ordering of assignments is immaterial, since the requirements on procedure call arguments ensures that the variables $\boldsymbol{y}$ are distinct from $\boldsymbol{x'}$.

To functionalise a procedure call $\mathsf{Call}(n, \boldsymbol{q})$, we build a new argument list $\boldsymbol{x'}$ of all the input arguments, then generate one assignment for each output argument. $\mathrm{In}_{\boldsymbol{q}}$ and $\mathrm{Out}_{\boldsymbol{q}}$ select the input and output arguments of an actual argument list with respect to a formal parameter list $\boldsymbol{q}$:

$$\mathrm{In}_{\boldsymbol{q}}(\boldsymbol{x}) = [x \mid \langle x, \langle m, t\rangle\rangle \leftarrow \langle \boldsymbol{x}, \boldsymbol{q}\rangle \text{ if } m \in \{\texttt{obs}, \texttt{upd}\}]$$
$$\mathrm{Out}_{\boldsymbol{q}}(\boldsymbol{x}) = [x \mid \langle x, \langle m, t\rangle\rangle \leftarrow \langle \boldsymbol{x}, \boldsymbol{q}\rangle \text{ if } m \in \{\texttt{out}, \texttt{upd}\}]$$

For example, given a `function int f(int)`, we may use the canonical proceduralisation (3) to obtain and use a `procedure p(obs int, out int)`. Since (3) only gives us single-output procedures, we will only get a single assignment for each call to p. Using (5) above, we can functionalise the following statements:

```
call p(5, x);          →      x = f(5);
call p(x, y);                 y = f(x);
```

Data-flow analysis may allow us to transform the code to `y = f(f(5))`, possibly eliminating the x if it is not needed.

3.4 Mutification of Whole Programs

Mutification has strict assumptions about the instructions it operates on, so we may need to perform instruction reordering and manipulation to make a program suitable for mutification:

Nested expressions must be broken up. This is done by moving a subexpression out of its containing expression and assigning it to a temporary variable. When functionalising a program, the reverse operation (expression inlining) can be applied to make as deeply nested expressions as possible, thus enabling easy application of high-level transformation rules.

Calls in control-flow statements must be moved outside. Mutification can't be applied directly to calls in conditions, `return`-statements and so on. Introducing a temporary, as above, we may replace the call with a reference to the variable (taking care to recompute the value of the variable for each loop iteration in the case of loops).

Instructions should be reordered to take advantage of mutification to a multi-valued procedure call. In general, if the instruction i_2 does not depend

on the result of i_1, and does not change variables in i_1, it can be moved in front of i_1. Otherwise, we must insert a dummy call with a throw-away result. For, example, given a procedure `p(obs int, obs int, out int, out int, out int)`:

```
y = f_2(a, b);          x = f_1(a, b);
x = f_1(a, b);    →     y = f_2(a, b);     →   int t;
                        int t = f_3(a, b);      call p(a, b, x, y, t);
```

Alternatively, we may apply *slicing* to create a specialised procedure variant without the unnecessary output.

Instructions may be made independent of each other by introducing a temporary. For example, if we want to move the second instruction in front of the first, we can store the value of y in a temporary variable:

```
x = f(y);          int t = y;          int t = y;
y = g(3);    →     x = f(t);     →     y = g(3);
                   y = g(3);           x = f(t);
```

Dig up your old compiler book! Instruction reordering and scheduling is well-known from compiler construction, and similar techniques can be applied here. For example, reordering assignments to take advantage of a multi-valued procedure call is not unlike reordering instructions to take advantage of parallel execution paths in a processor.

Many familiar optimisation techniques like constant propagation, value numbering, common sub-expression elimination, dead variable/code elimination and so on can readily be applied to Magnolia programs, either by first converting to a SSA (static single assignment) form, or by using simple data-flow analysis [10]. Pure functions and explicit information about input and outputs of procedures makes it a lot easier to apply optimisations.

Fig. 1 shows an example program before and after mutification, using the actual intermediate output of the mutification stage of the compiler.

4 Discussion

4.1 Working with Concepts and Axioms

The main benefit we have seen so far (apart from the somewhat fuzzily defined notational clarity) is in the relationship between code and specification, particularly with the concept feature.

- Most operations defined in a concept will have an algebraic-style declaration (with the exception of certain things like I/O). There is no need for deciding whether to define a functional-style interface, or an OO-style interface or an imperative-style interface.
- Axioms defined in concepts map directly to algebraic specifications. In C++, axioms for imperative-style concepts are typically written using the comma operator (sequential composition of expressions):

```
procedure fib(out int f, obs int n)
{
  if(n < 2)
    f = n;
  else
    f = fib(n-1) + fib(n-2);
}
```

```
procedure fib (out int f, obs int n) {
  bool a_0;
  call _<_(a_0, n, 2);
  if(a_0)
    f = n;
  else
    int c_0;
    c_0 = n;
    call _-_(c_0, 2);
    int b_0;
    call fibonacci::fib(b_0, c_0);
    int d_0;
    d_0 = n;
    call _-_(d_0, 1);
    call fibonacci::fib(f, d_0);
    call _+_(f, b_0);
}
```

```
procedure fib2(upd int n)
{
  if(n < 2)
    ;
  else
    n = fib2(n-1) + fib2(n-2);
}
```

```
procedure fib2 (upd int n) {
  bool e_0;
  call _<_(e_0, n, 2);
  if(e_0)
    ;
  else
    int f_0;
    f_0 = n;
    call _-_(f_0, 2);
    call fibonacci::fib2(f_0);
    call _-_(n, 1);
    call fibonacci::fib2(n);
    call _+_(n, f_0);
}
```

Fig. 1. Left: Two different variants of a recursive Fibonacci procedure. Right: Results of mutifying the two procedures (actual intermediate output from the compiler). Note that `fib2` – using an `upd` parameter – requires fewer temporaries than `fib`, which uses an `out`/`obs` combination. The notation `_+_(a,b)` is simply a desugared function-call variant of an operator call `a + b`.

```
axiom getset(A a, I i, E e) {
  setElt(a,i,e), getElt(a,i) == e;
}
```

- In most cases, there is a built-in well-defined mapping between procedure declarations and the function declarations used in a concept. As long as program code is written against the concept interfaces, axioms are easily related to program code. In C++, small wrappers are often required to translate between the declaration style in the concept and that of the implementation. This indirection makes it much harder to relate axioms to actual program code, for example for use as rewrite rules [11].

Axiom-based rewriting [2] is a convenient way to do high-level optimisation by using equational axioms as rewrite rules – similar to how algebraic laws are used to simplify arithmetic expressions. With code based on pure, well-behaved functions and no aliasing, this is very simple to implement, and rules can be applied without any of the complicated alias analysis that are part of modern compilers.

4.2 Limitations of Mutification

Mutification and functionalisation have some limitations. We are basically hiding imperative code behind an interface of pure functions, and if we are to do this without costly overhead, what we're hiding must be reasonably pure. This means that:

- Procedure results can only be influenced by input arguments.
- A procedure can have no other effect on program state than its effect on its output arguments.
- Objects used as function arguments must be *clonable* – i.e., it must be possible to save and restore the complete state of the object.

The two former limitations rule out procedures operating on global variables. The latter rules out things like stream I/O and user interaction.

Global variables are problematic because they interfere with reasoning – introducing unexpected dependencies or causing unexpected side-effects. This breaks the simple algebraic reasoning model that allows us to move code around and modify it without complicated analysis. Global variables can be handled in some cases by passing them as arguments, either explicitly or implicitly (having the compiler add global variables to declarations and calls). At some point we will reach an outermost procedure in the call tree which will have to either get all globals as arguments from the run-time system, or be allowed to access globals directly. If the global state is large, passing it around can be impractical, particularly since mutification will require that the state can be cloned when necessary.

The clonability requirement comes from the need to sometimes introduce temporaries while applying mutification, which is necessary to protect against aliasing, and also useful to avoid unnecessary recomputations. Some objects are however unclonable, e.g., because they represent some kind of interaction with the real world (reading and writing to a file or terminal, for example). Such objects are considered *impure*. They can only be used as `upd` arguments, and transformation on code involving impure objects must preserve the order of and conditions under which the objects are accessed. This is similar to how a C compiler would treat `volatile` variables in low-level code like device drivers.

Impure objects are best handled using procedures, though in principle we can mutify function calls involving impure objects as long as no copying is

required.

4.3 Performance

We have no performance data on Magnolia yet, but we have experimented with a simple form of mutification on the Sophus numerical software library [3,7], which is implemented in C++ in an algebraic coding style. Mutifying the code of the Seismod seismic simulation application, we saw a speedup factor of 1.8 (large data sets) to 2.0 (small data sets) compared to non-mutified algebraic-style code. Memory usage was reduced to 60%.

4.4 Related Work

Fluent languages [4] combine functional and imperative styles by separating the language into sub-languages, with *PROCEDUREs* which are fully imperative, *OBSERVERs*, which do not cause side-effects, *FUNCTIONs*, which do not cause and are not affected by side-effects, *PUREs* which are referentially transparent (no side-effects, and return the same value on every evaluation). The invariants are maintained by forbidding calls to subroutines with more relaxed restrictions. Our mutification, on the other hand, takes responsibility for protecting against harmful side-effects, allowing calls to procedures from functions.

Note that mutification assumes that no global variables are updated behind the scenes. Dealing with access to arbitrary parts of the store from within a function would complicate our method quite a lot. But since modern programming practises frown on global variables, this is not an intolerable restriction – and it is something the compiler can check for.

Mutification bears some resemblance to the translation to three-address code present in compilers [1]. Our approach is more high-level and general, dealing with arbitrary procedures instead of a predetermined set of assembly/intermediate-language instructions.

Copy elimination [5] is a method used in compilation functional languages to avoid unnecessary temporaries. By finding an appropriate target for each expression, evaluation can store the results directly in the place where it is needed, thus eliminating (at least some) intermediate copies.

Expression templates [12] attempt to avoid intermediates (and provide further optimisation) by using the template meta-programming facility in C++ to compile expressions (particularly large-size numerical expressions) directly into an assignment function.

5 Conclusion

We have presented a programming method and formal tool, which decouples the usage and implementation of operations. Through *functionalisation* and

mutification, we make imperative procedures callable as algebraic-style functions and provide a translation between code using function calls and code using procedure calls. This decoupling brings us the following benefits:

- Reuse—having a unified call style simplifies the interfacing of generic code with a wide range of implementations. Signature manipulation may also help integrate code from different sources. This is by no means a complete solution to the problem of reuse, but a small piece in the puzzle which may make things easier.
- Flexibility—multiple implementations with different performance characteristics can be accessed through the same interface.
- Notational clarity of functional style – algebraic notation is similar to mathematics and well suited to express mathematical problems.
- Link to axioms and algebraic specification. The algebraic notation can be directly related to the notation used in specifications, thus bridging the gap between practical programming and formal specifications. This enables the use of axioms for high-level optimisations and for automated testing [2].
- The procedural imperative style with in situ updating of variables provide better space and time efficiency, particularly for large data structures.

The notational and reasoning benefits are similar to what is offered by functional programming languages, without requiring immutable variables.

Our initial experiments with mutification were done on C++, because the excellent, high-performance compilers available for C++ make it a good choice for performance-critical code. Languages like C++ are however notoriously difficult to analyse and process by tools. The idea behind Magnolia is to cut away the parts of C++ that are difficult to process or that interfere with our ability to reason about the code, and then add the features necessary to support our method. The Magnolia compiler will produce C++ code, allowing us to leverage both good compilers and good programming methodology.

This paper expands our earlier work [3] on mutification by (1) allowing the imperative forms to update an arbitrary number of arguments (previously limited to one), (2) allowing functionalisation of procedure calls, (3) providing a formal treatment of functionalisation and mutification, and (4) doing this independently of the numerical application domain which was the core of [3]. Functionalisation and mutification was originally motivated by code clarity and efficiency concerns, with the benefits to generic programming becoming apparent later on.

More work remains on determining the performance improvement that can be expected when using imperative vs. algebraic styles, and the productivity improvement that can be expected when using algebraic vs. imperative style. Both these aspects are software engineering considerations. Further work on formally proving the effectiveness and correctness of mutification is also needed.

A prototype implementation of Magnolia, supporting functionalisation of declarations and mutification of calls, is available at http://magnolia-lang.org.

References

[1] Aho, A. V., R. Sethi and J. D. Ullman, "Compilers — Principles, Techniques, and Tools," Addison-Wesley, 1986.

[2] Bagge, A. H. and M. Haveraaen, *Axiom-based transformations: Optimisation and testing*, in: J. Vinju and A. Johnstone, editors, *Eighth Workshop on Language Descriptions, Tools and Applications (LDTA 2008)*, Electronic Notes in Theoretical Computer Science (2008).

[3] Dinesh, T., M. Haveraaen and J. Heering, *An algebraic programming style for numerical software and its optimization*, Scientific Programming **8** (2000), pp. 247–259.

[4] Gifford, D. K. and J. M. Lucassen, *Integrating functional and imperative programming*, in: *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming* (1986), pp. 28–38.

[5] Gopinath, K. and J. L. Hennessy, *Copy elimination in functional languages*, in: *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1989), pp. 303–314.

[6] Guttag, J. and J. Horning, *Report on the Larch shared language*, Science of Computer Programming **6** (1986), pp. 103–134.

[7] Haveraaen, M., H. A. Friis and T. A. Johansen, *Formal software engineering for computational modelling*, Nordic Journal of Computing **6** (1999), pp. 241–270.

[8] Jones, S. P., A. Tolmach and T. Hoare, *Playing by the rules: Rewriting as a practical optimisation technique in GHC*, in: R. Hinze, editor, *2001 Haskell Workshop*, Firenze, Italy, 2001.

[9] Mosses, P. D., editor, "CASL Reference Manual: The Complete Documentation of the Common Algebraic Specification Language," LNCS **2960**, Springer-Verlag, 2004.

[10] Olmos, K. and E. Visser, *Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules*, in: R. Bodik, editor, *14th International Conference on Compiler Construction (CC'05)*, Lecture Notes in Computer Science **3443** (2005), pp. 204–220.

[11] Tang, X. and J. Järvi, *Concept-based optimization*, in: *Proceedings of the ACM SIGPLAN Symposium on Library-Centric Software Design (LCSD'07)* (2007).

[12] Veldhuizen, T. L., *Expression templates*, C++ Report **7** (1995), pp. 26–31, reprinted in C++ Gems, ed. Stanley Lippman.