Manuel Serrano
Jurriaan Hage (Eds.)

LNCS 9547

Trends in Functional Programming

16th International Symposium, TFP 2015
Sophia Antipolis, France, June 3–5, 2015
Revised Selected Papers

Lecture Notes in Computer Science 9547

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

More information about this series at http://www.springer.com/series/7407

Manuel Serrano · Jurriaan Hage (Eds.)

Trends in Functional Programming

16th International Symposium, TFP 2015
Sophia Antipolis, France, June 3–5, 2015
Revised Selected Papers

Springer

Editors
Manuel Serrano
INRIA Sophia Antipolis
Sophia Antipolis
France

Jurriaan Hage
Department of Information and Computing Science
Utrecht University
Utrecht
The Netherlands

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-39109-0 ISBN 978-3-319-39110-6 (eBook)
DOI 10.1007/978-3-319-39110-6

Library of Congress Control Number: 2016939385

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG Switzerland

Preface

This volume contains a selection of the papers presented at TFP 2015: the Symposium on Trends in Function Programming 2015, held during June 3–5, 2015, in Sophia Antipolis, France.

TFP is an international forum for researchers with interests in all aspects of functional programming, taking a broad view of current and future trends in the area. It aspires to be a lively environment for presenting the latest research results and other contributions, described in draft papers submitted prior to the symposium. For the symposium, the Program Committee chair verified that these drafts were within the scope of TFP. Submissions appearing in the draft proceedings are not considered as peer-reviewed publications.

The TFP 2015 program consisted of two invited talks, one tutorial, and 26 presentations. The invited talks were given by Laurence Rideau (Inria, France) on "Engineering Mathematics: The Odd Order Theorem Proof," and Florian Loitsch (Google, Denmark) on "Dart, An Introduction." The tutorial was given by Florian Loitsch on "Programming with Dart." The 26 presentations led to a total of 24 full papers submitted to the formal post-refereeing process. Each submission was reviewed by at least three reviewers. The Program Committee selected eight papers, which are included in these proceedings.

We are grateful to everyone at Inria for their help in preparing and organizing TFP 2015, in particular Agnes Cortell and Nathalie Bellesso. We gratefully acknowledge the financial support of Inria, which allowed us to maintain low registration costs. We also gratefully acknowledge the assistance of the TFP 2015 Program Committee and the TFP Steering Committee for their advice while organizing the symposium.

Februrary 2016

Manuel Serrano
Jurriaan Hage

Organization

Program Committee

Edwin Brady	University of St. Andrews, UK
Olaf Chitil	University of Kent, UK
Marc Feeley	Université de Montréal, Canada
Jean-Christophe Filliâtre	Université Paris Sud Orsay, France
Lars-Åke Fredlund	Universidad Politécnica de Madrid, Spain
Thomas Gazagnaire	University of Cambrige, UK
Andy Gill	University of Kansas, USA
Jurriaan Hage	Utrecht University, The Netherlands
Daan Leijen	Microsoft, USA
Sam Lindley	The University of Edinburgh, UK
Wolfgang De Meuter	Vrije Universiteit Brussel, Belgium
Michel Mauny	Ensta ParisTech, France
Marco T. Morazan	Seton Hall University, USA
Scott Owens	University of Kent, UK
Tomas Petricek	University of Cambridge, UK
Rinus Plasmeijer	University of Nijmegen, The Netherlands
Colin Runciman	University of York, UK
Neil Sculthorpe	Swansea University, UK
Manuel Serrano (Chair)	Inria, France
Janis Voigtländer	University of Bonn, Germany

Additional Reviewers

Simon Fowler
Mark Grebe
J. Garrett Morris
Fernando Rubio

Sponsoring Institutions

Inria Méditerranée

Contents

Lightweight Higher-Order Rewriting in Haskell

Emil Axelsson(✉) and Andrea Vezzosi

Chalmers University of Technology, Gothenburg, Sweden
emax@chalmers.se

Abstract. We present a generic Haskell library for expressing rewrite rules with a safe treatment of variables and binders. Both sides of the rules are written as typed EDSL expressions, which leads to syntactically appealing rules and hides the underlying term representation. Matching is defined as an instance of Miller's higher-order pattern unification and has the same complexity as first-order matching. The restrictions of pattern unification are captured in the types of the library, and we show by example that the library is capable of expressing useful simplifications that might be used in a compiler.

1 Introduction

Work on embedded domain-specific languages (EDSLs) has taught us many useful techniques for constructing terms: smart constructors for hiding the underlying representation of expressions, higher-order functions to represent constructs that introduce local variables, phantom types to give a typed interface to an untyped representation, etc. Unfortunately, these techniques are only applicable to term construction, not to pattern matching. Pattern matching is needed to examine expressions; for example in transformations, interpretation or compilation.

So, although EDSL *users* have a very nice interface for constructing expressions, EDSL *implementors* are confined to working with the underlying representation. This can lead to several problems:

- Type safety: If the representation is untyped, it is easy to cause type errors when transforming expressions.
- Verbosity: The representation may be inconvenient to work with, especially if it is based on generic encodings, such as compositional data types [4].
- Scoping: When transforming expressions with binders, it is easy to cause variables to escape their scope.

Although solutions or partial solutions exist for all of these problems, we are not aware of any solution in Haskell that handles all of them at once. This paper addresses all three problems in a single generic Haskell library for rewrite rules. Our library is also efficient: the complexity of rule application is determined only by the size of the rule. However, the library is restricted to plain rewrite rules – it cannot be used to define arbitrary functions on expressions.

M. Serrano and J. Hage (Eds.): TFP 2015, LNCS 9547, pp. 1–21, 2016.
DOI: 10.1007/978-3-319-39110-6_1

1.1 Running Example

As our running example, we will use the for-loop in the Feldspar EDSL [3]. Feldspar is a Haskell EDSL for signal processing and numeric computations. It supports common functional programming idioms, such as `map` and `fold`, and generates high-performance C code from such programs.

One of the more low-level constructs in Feldspar is `forLoop`:

```
forLoop :: Data Int → Data s → (Data Int → Data s → Data s) → Data s
```

`Data` is Feldspar's expression type which is parameterized by the type of the value the expression computes. The first argument to `forLoop` is the number of iterations; the second argument is the initial state; the third argument is the body which computes the next state given the loop index and the current state; the result is the final state of the loop.

We are interested in expressing the following simplification rules for `forLoop`:

- If the number of iterations is 0, the result is the initial state.
- If the body always returns the previous state, the result is the initial state.
- If the body does not refer to the previous state, it is enough to run the last iteration of the loop.

Furthermore, we would like to express these rules in a way that

- is independent of the representation of `Data`,
- does not allow accidentally changing the type of the expression,
- does not require looking at concrete variable identifiers,
- does not allow creating an ill-scoped expression.

To illustrate the problem, we try to express the rewrite rules as cases in a Haskell function. Assuming `Data` is a simple recursive data type, with constructors for lambda abstraction, variables, for-loops, etc., we might express the first two rules as follows:

```
simplify (ForLoop (Int 0) init _)                          = init
simplify (ForLoop _ init (Lam i (Lam s (Var s')))) | s == s' = init
```

Even though the definition looks quite readable, it violates most of our requirements on rewrite rules: it leaks the representation of `Data`, does not guarantee well-typedness, and involves comparing variable names.

The third rule is trickier. We want to rewrite an expression of the form

```
forLoop len init (λi s → body)
```

to

```
cond (len==0) init body'
```

where `body'` is `body` with `len-1` substituted for `i` and provided that `s` does not occur freely in `body`. The object-level function `cond` is used to return `init` when the length is 0 and otherwise return `body'`.

Trying to express this rule as a case in `simplify` reveals an additional problem of this style of rewriting: it is possible for `body` to contain free variables. In order

to prevent these variables from escaping their scope, either we need to check for their absence or we need to substitute for these variables on the right hand side. In the case of the third for-loop rule, we need to check that `s` does not occur freely in `body` and we need to substitute an expression for `i` on the right hand side. Either of these actions is very easy to forget.

As a preview of our solution, here is the third `forLoop` rule expressed using our library:

```
rule_for3 len init body =
    forLoop (meta len) (meta init) (λi s → body -$- i)
      ⟹
    cond (meta len === 0) (meta init) (body -$- (meta len - 1))
```

Note the use of Haskell's α-abstraction to give the pattern for the loop body. In addition to being quite close to the desired syntax, the rule is also guaranteed to be well-typed and well-scoped.

1.2 Overview of the Paper

Section 2 presents the basics of our rewriting library restricted to first-order matching. Section 3 revisits the general problem of higher-order matching and gives a simple algorithm for matching and rewriting based on Miller's pattern unification. Section 4 shows how our library can be extended to support higher-order matching. Section 5 demonstrates the library using a simple version of the Feldspar EDSL.

The rewriting library is available on Hackage.[1] The code makes use of many Haskell extensions, including `TypeFamilies`, `GADTs`, `DerivingFunctor`, etc. Consult the GHC documentation[2] for more information on these extensions.

2 A Generic Library for Rewrite Rules

In this section we show a first-order version of our library. The higher-order version in Sect. 4 is mostly an extension of the definitions in this section. Only the representation of meta-variables needs to be modified.

A rule is a pair of a left hand side (LHS) and a right hand side (RHS):

```
data Rule lhs rhs where
  Rule :: lhs a → rhs a → Rule lhs rhs
```

The parameters `lhs` and `rhs` are representations of the left and right hand sides of the rule. These representations are parameterized by the type of the corresponding expression, just like `Data` in Sect. 1.1. The type parameter is existentially quantified, and the only thing we care about is that `lhs` and `rhs` have the same type parameter.

[1] http://hackage.haskell.org/package/ho-rewriting-0.2.
[2] https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghc-language-features.html.

Rather than using fixed types for `lhs` and `rhs`, we will express our rules using type classes. This will allow us to use many of the same functions to express both sides of a rule, even if the two sides will in the end have slightly different representations. Using type classes also allows us to extend the rule language with new constructs simply by adding additional class constraints. Essentially, we regard `lhs` and `rhs` as languages in the final tagless style [6].

The first classes we introduce are for meta-variables and wildcards:

```
class MetaVar r where
  type MetaRep r :: * → *
  meta :: MetaRep r a → r a

class WildCard r where
  __ :: r a
```

The function `meta` introduces a meta-variable given a representation for it. The reason for making the `MetaRep` an associated type is to be able to disallow inspection of the representation of meta-variables. As long as we keep `r` abstract, `MetaRep r` will also be an abstract type. The method _ (double underscore) of the `WildCard` class constructs a pattern that matches any term. As we will see, our implementation only allows wildcards on the LHS of a rule.

Next, we introduce a convenient short hand for rules:

```
(⟹) :: lhs a → rhs a → Rule lhs rhs
(⟹) = Rule

infix 1 ⟹
```

Interestingly, we now have all the machinery we need to start expressing some rules for numeric operations. Each rule is given as a Haskell definition that takes the necessary meta-variables as arguments:[3]

```
-- 0 + X  ⟹  X
rule_add x = 0 + meta x  ⟹  meta x

-- X - X  ⟹  0
rule_sub x = meta x - meta x  ⟹  0

-- 0 * _  ⟹  0
rule_mul = 0 * __  ⟹  (0 :: _ Int)
```

How is it that we can already write rules about numeric operations without even having given a representation for the LHS and RHS of rules? Looking at the inferred type of `rule_add` tells us what is going on:

```
rule_add :: (MetaVar lhs, MetaVar rhs, MetaRep lhs ~ MetaRep rhs, Num (lhs a))
         ⇒ MetaRep lhs a → Rule lhs rhs
```

[3] Note the *partial* type annotation `(... :: _ Int)` in `rule_mul`. It is used to constrain the type parameter of the RHS without saying anything about the representation of the RHS. Partial type signatures require the recent `PartialTypeSignatures` extension to GHC. However, this extension is not strictly needed: an equivalent formulation of the RHS would be `(id :: r Int → r Int) 0`.

Since the rules are expressed entirely using type class operations (including those of the `Num` class), the type is polymorphic in `lhs` and `rhs`. But we see a number of constraints due to the way the operations are used. The constraints tell us that both sides have to support meta-variables, and whatever the representation of meta-variables is, it must be the same on both sides. Furthermore, `lhs` has to have a `Num` instance. The type parameter `a` to `MetaRep r` ensures that meta-variables are used at the same type if they occur multiple times a rule.

The partial type annotation in `rule_mul` is used to fix the type of the rule (i.e. the parameter to `lhs` and `rhs`). It is needed because `rule_mul` does not take a meta-variable identifier as argument, so the numeric type does not show up in the type of the rule (except in the context):

```
rule_mul :: (WildCard lhs, Num (lhs Int), Num (rhs Int)) ⇒ Rule lhs rhs
```

Of course, much more work is needed before we can actually do something with the above rules, but the rules themselves will not need any modifications. They can be used with our library as they stand.

2.1 Representation of Terms and Patterns

We need different restrictions on the different representations in our library:

- Meta-variables are allowed in rules, but not in the terms being rewritten.
- Wildcards are only allowed on the LHS, not on the RHS of rules.

However, all constructs of the object language should be available to use in the rules.

In order to maintain these restrictions while allowing maximal sharing between the representations, we make use of Data Types à la Carte [21]. The basic idea is to use a standard fixed-point data type parameterized by a base functor:

```
data Term f = Term { unTerm :: f (Term f) }
```

`Term` is a recursive data type where each node is a value of the base functor `f`. By using different `f` types, we can represent terms of different signatures.

Sharing between different representations is achieved by expressing the base functor as a co-product of smaller functors. Co-products are formed by the `:+:` type, which can be seen as a higher-kinded version of the `Either` type:

```
data (f :+: g) a = Inl (f a)
                 | Inr (g a)

infixr :+:
```

For example, given two functors representing numeric and logic operations

```
data NUM a                    data LOGIC a
    = Int Int                     = Bool Bool
    | Add a a                     | Not a
    | Sub a a                     | And a a
    | Mul a a                     | Equal a a
  deriving (Functor)              | Cond a a a
                                deriving (Functor)
```

we can form expressions of numeric and logic operations by using their coproduct as the base functor of a `Term`:

```
type Exp = Term (NUM :+: LOGIC)
```

The concrete representations for left and right hand sides of rules are defined as follows:

```
newtype LHS f a = LHS { unLHS :: Term (WILD :+: META :+: f) }
newtype RHS f a = RHS { unRHS :: Term (META :+: f) }

data WILD a = WildCard  deriving Functor
data META a = Meta Name deriving Functor

type Name = Int
```

Both `LHS` and `RHS` are parameterized on a base functor `f` representing the signature of the language the rules operate on. `LHS` extends `f` with meta-variables and wildcards while `RHS` only extends `f` with meta-variables. Both `LHS` and `RHS` have a phantom type parameter `a` which denotes the type of the corresponding term. It is used to ensure that only well-typed left and right hand sides can be constructed.

We can now make instances of the classes introduced earlier:

```
instance WildCard (LHS f) where
  _ = LHS $ Term $ Inl WildCard

instance MetaVar (LHS f) where
  type MetaRep (LHS f) = META
  meta = LHS . Term . Inr . Inl . castMETA

instance MetaVar (RHS f) where
  type MetaRep (RHS f) = META
  meta = RHS . Term . Inl . castMETA
```

Note that `META` is used in two roles here: (1) as the constructor for meta-variables in `LHS` and `RHS`, and (2) as the concrete instance of `MetaRep`. The function `castMETA` is used to convert between these two roles:

```
castMETA :: META a → META b
castMETA (Meta v) = Meta v
```

For example, in the instance `MetaVar (LHS f)`, we have `meta :: META a → LHS f a` and then `castMETA` is used at the concrete type

```
castMETA :: META a → META (Term (WILD :+: (META :+: f)))
```

Our library makes use of the Compdata package [4] for the implementation of `Term` and `:+:`. Compdata is a Haskell library based on Data Types à la Carte, and it provides many utilities for working with representations based on `Term`.

2.2 Matching and Rewriting

We will now give a formal definition of the rewriting algorithm used in our library. The following grammar defines terms and rules:

SYMBOLS	f, g	a set of symbols with associated arities
META-VARIABLES	M	
TERMS	t	$::= f\ \vec{t}$
LHS	l	$::= f\ \vec{l} \mid M \mid \_$
RHS	r	$::= f\ \vec{r} \mid M$
RULES	ρ	$::= l \implies r$

A term t is a tree where each node has a symbol f and zero or more sub-trees. A left hand side l is a term extended with meta-variables and wildcards, and a right hand side r is a term that is only extended with meta-variables.

The first-order version of our library is based on standard syntactic rewriting, as defined in Fig. 1. The matching relation $l \stackrel{?}{=} t \rightsquigarrow \sigma$ defines how matching a term t against a pattern l results in a list σ of mappings from meta-variables to sub-terms. Wildcards and meta-variables match any term, with the difference that matching against a meta-variable results in a mapping in the substitution. For symbols, matching is done recursively for the children, and the resulting substitutions are concatenated.

Rewriting is defined as matching a term against the LHS and applying the corresponding substitution to the RHS. We use $[\![\sigma]\!]r$ to denote application of a substitution σ to r. Since we allow non-linear patterns, where the same meta-variable occurs more than once, we also have to check that the substitution

$$\frac{}{\_ \stackrel{?}{=} t \rightsquigarrow []}\ \text{WILD} \qquad \frac{}{M \stackrel{?}{=} t \rightsquigarrow [M \mapsto t]}\ \text{META}$$

$$\frac{f = g \quad l_1 \stackrel{?}{=} t_1 \rightsquigarrow \sigma_1 \quad \ldots \quad l_n \stackrel{?}{=} t_n \rightsquigarrow \sigma_n}{f\ l_1 \ldots l_n \stackrel{?}{=} g\ t_1 \ldots t_n \rightsquigarrow concat(\sigma_1 \ldots \sigma_n)}\ \text{SYMBOL}$$

$$rewrite(l \implies r, t) = [\![\sigma]\!]r \quad \text{where} \quad l \stackrel{?}{=} t \rightsquigarrow \sigma,\ consistent(\sigma)$$

Fig. 1. First-order matching and rewriting.

obtained from matching is consistent; i.e. that each given meta-variable only maps to equal terms.

The corresponding functions in our library are

```
type Subst f = [(Name, Term f)]  -- Substitution

match :: (Functor f, Foldable f, EqF f)
      ⇒ LHS f a → Term f → Maybe (Subst f)


substitute :: Traversable f ⇒ Subst f → RHS f a → Maybe (Term f)
```

The `match` function succeeds if and only if the `LHS` matches the term and all occurrences of a given meta-variable are matched against equal terms. The `substitute` function succeeds if and only if each meta-variable in the RHS has a mapping in the substitution. The `EqF` class comes from the Compdata package, and is used for comparing symbols.

Combining `match` and `substitute` gives us the `rewrite` function:

```
rewrite :: (Traversable f, EqF f)
        ⇒ Rule (LHS f) (RHS f) → Term f → Maybe (Term f)
rewrite (Rule lhs rhs) t = do
    subst ← match lhs t
    substitute subst rhs
```

When working with lists of rewrite rules, we are often interested in trying the rules in sequence and picking the first one that applies. That is the purpose of `applyFirst`:

```
applyFirst :: (Traversable f, EqF f)
           ⇒ [Rule (LHS f) (RHS f)] → Term f → Term f
applyFirst rs t = case [t' | rule ← rs, Just t' ← [rewrite rule t]] of
                       t':_ → t'
                       _    → t
```

If no rule matches, `applyFirst` returns the original term.

Another strategy is to rewrite each node in a term from bottom to top:

```
bottomUp :: Functor f ⇒ (Term f → Term f) → Term f → Term f
bottomUp rew = rew . Term . fmap (bottomUp rew) . unTerm
```

The first argument to `bottomUp` is the node rewriter. Since each node is a functor value, we use `fmap` to recursively transform all children. Then we apply the node rewriter to the resulting term.

A top-down strategy is defined in a similar way; just apply the rewrite before the recursive call:

```
topDown :: Functor f ⇒ (Term f → Term f) → Term f → Term f
topDown rew = Term . fmap (topDown rew) . unTerm . rew
```

Typically, one is interested in combinations such as `bottomUp (applyFirst rs)`, which applies the first matching rule in the list `rs` to each node in a term.

3 Higher-Order Rewriting

The library presented in Sect. 2 works well for first-order rules, such as `rule_add` from earlier. But in order to express simplification rules for the for-loop in Sect. 1.1, we need to extend the library and the rewriting algorithm with support for higher-order terms and rules.

The matching algorithm from Fig. 1 is purely *syntactic*. It obeys the following property, where $=$ is syntactic equality:[4]

$$l \stackrel{?}{=} t \rightsquigarrow \sigma \quad \Rightarrow \quad [\![\sigma]\!]l = t$$

Higher-order matching [11,22], on the other hand, obeys the following *semantic* property, where $t \equiv_{\alpha,\beta,\eta} u$ means that t and u reduce to the same term up to α-renaming:

$$l \stackrel{?}{=} t \rightsquigarrow \sigma \quad \Rightarrow \quad [\![\sigma]\!]l \equiv_{\alpha,\beta,\eta} t$$

Substitution in the higher-order case must be *capture-avoiding*.

If we extend our rule language to higher-order rules, the third rule of the for-loop in Sect. 1.1 can be defined as follows:

$$forLoop\ \textsc{len}\ \textsc{init}\ (\lambda i.\lambda s.\ \textsc{body}\ i) \implies cond\ (eq\ \textsc{len}\ 0)\ \textsc{init}\ (\textsc{body}\ (sub\ \textsc{len}\ 1))$$

We use the convention to write meta-variables using SMALLCAPS. The symbols *forLoop*, *cond*, *eq* and *sub* represent for-loops, conditions, equality and subtraction, respectively. We also treat numeric literals as predefined symbols.

Using normal syntactic matching semantics, the above rule would only match a for-loop whose body binds exactly the variables i and s, and where some expression is immediately applied to i inside the abstraction. However, using higher-order matching semantics, the pattern $\lambda i.\lambda s.\ \textsc{body}\ i$ matches any expression with two enclosing λ-abstractions and a body that only refers to the first bound variable.

As an example, we match the term t_1 against the pattern l_1 defined as follows:

$$\begin{aligned} t_1 &= forLoop\ 10\ 0\ (\lambda x.\lambda y.\ sub\ x\ 2) \\ l_1 &= forLoop\ \textsc{len}\ \textsc{init}\ (\lambda i.\lambda s.\ \textsc{body}\ i) \end{aligned}$$

Despite the fact that $sub\ x\ 2$ is not an immediate application to x, the pattern matches, and results in the substitution

$$\begin{aligned} \sigma_1 = [\ &\textsc{len} \mapsto 10 \\ , &\textsc{init} \mapsto 0 \\ , &\textsc{body} \mapsto \lambda z.\ sub\ z\ 2\ ] \end{aligned}$$

We check the result by applying σ_1 to l_1 which gives a result equivalent to t_1:

$$[\![\sigma_1]\!]l_1 = forLoop\ 10\ 0\ (\lambda i.\lambda s.\ (\lambda z.\ sub\ z\ 2)\ i) \equiv_{\alpha,\beta,\eta} t_1$$

[4] The property is almost true: it holds if we replace all wildcards in l with unique meta-variables.

An alternative to introducing a fresh variable z for BODY is to reuse the existing variable name x. That would give the following substitution instead:

$$\sigma_1 \;=\; [\; \ldots,\; \text{BODY} \mapsto \lambda x.\; sub\; x\; 2\;]$$

This result is just as valid as the previous one, and it has the advantage that the body $sub\ x\ 2$ does not need to be renamed.

An implicit side condition in higher-order matching is that the resulting substitution is not allowed to contain free variables that were not free in the original term. For example, the following term does not match l_1:

$$t_2 \;=\; forLoop\ 10\ 0\ (\lambda x.\lambda y.\; sub\; x\; y)$$

An attempt at a solution might be

$$\sigma_2 \;=\; [\; \ldots,\; \text{BODY} \mapsto \lambda x.\; sub\; x\; s\;]$$

This solution has s as a free variable. However, $[\![\sigma_2]\!]l_1$ is not equivalent to t_2, because substitution is defined to be capture-avoiding.

If we want to allow s to occur in the body, we need to declare that by listing s as one of the arguments to BODY:

$$l_2 \;=\; forLoop\ \text{LEN INIT}\ (\lambda i.\lambda s.\ \text{BODY}\ i\ s)$$

Matching t_2 against this pattern results in the substitution

$$\sigma_3 \;=\; [\; \ldots,\; \text{BODY} \mapsto \lambda x.\lambda y.\; sub\; x\; y\;]$$

for which it holds that $[\![\sigma_3]\!]l_2$ is equivalent to t_2.

3.1 Tractability

Higher-order matching is an instance of higher-order unification, with the difference that the latter permits meta-variables on both sides of the $\stackrel{?}{=}$ relation. Higher-order unification is undecidable in general [10]. Higher-order matching is decidable, but its complexity class is at least NP-complete for second-order problems and upwards [22].

Miller identified a fragment for which unification is efficient, namely when each meta-variable is applied only to distinct object-language variables [12]. Note that l_1 and l_2 from before fall under this category, because BODY is only applied to the object-language variables i and s. This restriction of the general problem is called the *pattern fragment*. The term "pattern" refers to the list of object-language variables that a meta-variable is applied to, and should not be confused with its use in the term "pattern matching".

Symbols	f	a set of symbols with associated arities	
Object variables	v		
Atoms	a, b	$::=$	$v \mid f$
Meta-variables	M		
Terms	t	$::=$	$a\ \vec{t} \mid \lambda v.t \quad (\mid t\ \vec{t})$
LHS	l	$::=$	$a\ \vec{l} \mid \lambda v.l \mid M\ \vec{v} \mid \_$
RHS	r	$::=$	$a\ \vec{r} \mid \lambda v.r \mid M\ \vec{r}$
Rules	ρ	$::=$	$l \implies r$

Fig. 2. Grammar for higher-order terms and rewrite rules in the pattern fragment.

$$\frac{}{\_ \stackrel{?}{=} t \rightsquigarrow []}\ \text{WILD} \qquad \frac{l \stackrel{?}{=} t \rightsquigarrow \sigma}{\lambda v.l \stackrel{?}{=} \lambda v.t \rightsquigarrow \sigma}\ \text{LAM}$$

$$\frac{a = b \quad l_1 \stackrel{?}{=} t_1 \rightsquigarrow \sigma_1 \quad \ldots \quad l_n \stackrel{?}{=} t_n \rightsquigarrow \sigma_n}{a\ l_1 \ldots l_n \stackrel{?}{=} b\ t_1 \ldots t_n \rightsquigarrow concat(\sigma_1 \ldots \sigma_n)}\ \text{ATOM}$$

$$\frac{freeVars(\lambda v_1 \ldots \lambda v_n.t) = \emptyset}{M\ v_1 \ldots v_n \stackrel{?}{=} t \rightsquigarrow [M \mapsto \lambda v_1 \ldots \lambda v_n.t]}\ \text{META}$$

Fig. 3. Simplified higher-order matching for the pattern fragment.

3.2 Rewriting Based on Pattern Unification

Matching for the pattern fragment can be done as a lightweight extension to the first-order algorithm presented in Sect. 2.2.

Figure 2 shows the previous grammar extended with object-language variables and λ-abstraction. We ensure that terms and rules are in β-short normal form by making use of the so-called spine formulation [7] which disallows application of λ-abstractions. We do however allow general applications in the result after rewriting, which is why the production $t\ \vec{t}$ for terms is put in parentheses. On the LHS, meta-variables can only be applied to object-language variables, while this restriction is not needed on the RHS.

A simplified higher-order matching algorithm is defined in Fig. 3. The SYM rule has been replaced with the ATOM rule, which covers both symbols and object-language variables. λ-abstractions are matched structurally. Meta-variables are matched simply by turning the list of arguments into a number of λ-abstractions, as we did previously in the for-loop example. Like in that example, we also reuse the names $v_1 \ldots v_n$ in the lambda abstractions, which avoids having to rename variables in t.

The given algorithm is a bit simplified for presentation purposes:

- It does not deal with α-renaming.
- It does not allow any free variables in the substitution. As mentioned earlier, we can allow free variables if they were already free in the original term.

– It assumes that λ is always matched against λ. For example, the term $\lambda v.f\ v$ will not match its η-reduced form f, as it should.

The implementation in our library deals correctly with α-renaming and free variables. The simplest way to deal with η conversion is to always η-expand subexpressions of function type to get terms in η-long normal form. Our matching algorithm currently does not do this; however, it is possible to define the user interface in such a way that partially applied atoms do not occur in practice. We will see how that is done in Sect. 5.

Once higher-order matching has been defined, higher-order rewriting is defined analogously to the *rewrite* function in Fig. 1. It should be noted that when substituting for meta-variables on the RHS, we may create β-redexes for meta-variables that have arguments. In our implementation, it is possible to choose whether to reduce those redexes immediately or leave them for later.

Matching according to the rules in Fig. 3 is efficient in the sense that the number of recursive steps is bounded by the size of the pattern. The only possible source of inefficiency is the use of $freeVars$ in the META rule. Our implementation avoids traversing the whole term when checking the free variables simply by caching the set of free variables for each node in a term. The result is that the complexity of rule application is determined only by the size of the rule – just like for first-order matching.

3.3 Most General Solutions

There is one aspect of Miller's pattern restriction that we do not enforce: meta-variables must only be applied to *distinct* object-language variables. This restriction is needed to ensure the existence of a most general unifier. The main reason we do not enforce it is that it is hard to capture this particular restriction in the types of the library.

For example, when matching $\lambda x.\ sub\ x\ 2$ against $\lambda y.$ BODY $y\ y$, there are two possible solutions: BODY $\mapsto \lambda a.\lambda x.\ sub\ x\ 2$ and BODY $\mapsto \lambda x.\lambda a.\ sub\ x\ 2$. Our implementation will blindly give the result BODY $\mapsto \lambda x.\lambda x.\ sub\ x\ 2$, which is equivalent to the first solution. There is nothing wrong with either solution; the only problem is that picking one instead of the other is a bit arbitrary.

To avoid this problem, the library user must make sure to only apply meta-variables to distinct object-language variables.

4 Extending the Library to Higher-Order Rewriting

We will now show how to extend the first-order library from Sect. 2 to higher-order rewriting. LHS and RHS in Fig. 2 permit application of meta-variables to objects of different kinds. LHS only allows application to object-language variables, while RHS allows application to arbitrary terms. We reconcile these different requirements using the type `MetaExp` which represents meta-variables applied to a number of arguments:

```
data MetaExp (r :: * → *) a where
  MVar :: MetaRep r a → MetaExp r a
  MApp :: MetaExp r (a → b) → MetaArg r a → MetaExp r b

type family MetaRep (r :: * → *) :: * → *
type family MetaArg (r :: * → *) :: * → *
```

The representation of the meta-variable is given by the type family `MetaRep` (corresponding to the associated type of the same name in Sect. 2), and the representation of the arguments is given by `MetaArg`. By using different `MetaArg` representations, we can enforce different requirements for meta-variable application in the LHS and RHS.

We introduce yet another type family which gives an abstract representation of object-language variables:

```
type family Var (r :: * → *) :: * → *
```

We can now give the following `MetaArg` instances for `LHS` and `RHS`:

```
type instance MetaArg (LHS f) = Var (LHS f)
type instance MetaArg (RHS f) = RHS f
```

The first instance ensures that meta-variables can only be applied to object-language variables on the LHS, while the second instance permits arbitrary terms as meta-variable arguments on the RHS.

We redefine the `MetaVar` class with a single method that constructs an expression from a `MetaExp` value:

```
class MetaVar r where
  metaExp :: MetaExp r a → r a

instance MetaVar (LHS f)
  -- see library source for details

instance MetaVar (RHS f)
  -- see library source for details
```

Introducing meta-variables using `MVar`, `MApp` and `metaExp` is quite cumbersome, so we provide a number of helper functions:

```
meta :: MetaVar r ⇒ MetaRep r a → r a
meta = metaExp . MVar

($$)  ::              MetaExp r (a → b) → MetaArg r a → MetaExp r b
($-)  :: MetaVar r ⇒ MetaExp r (a → b) → MetaArg r a → r b
(-$)  ::              MetaRep r (a → b) → MetaArg r a → MetaExp r b
(-$-) :: MetaVar r ⇒ MetaRep r (a → b) → MetaArg r a → r b

($$)    = MApp
f  $- a = metaExp (MApp f a)
f -$  a = MApp (MVar f) a
f -$- a = metaExp (MApp (MVar f) a)

infixl 2 $$, $-, -$, -$-
```

The function `meta` has the same type as in Sect. 2, and it introduces a meta-variable without any arguments. For meta-variables with arguments, we use the different application operators:

- `-$-` is used when there is only one argument.
- `-$` is used for the first argument when there are more than one argument.
- `$-` is used for the last argument when there are more than one argument.
- `$$` is used for used for any but the first and last arguments.

As an example, assume we have two meta-variables and two object-language variables of the following types (for some base functor `F`):

```
m1 :: MetaRep (LHS F) Int
m2 :: MetaRep (LHS F) (Int → Char → Bool)
v1 :: Var (LHS F) Int
v2 :: Var (LHS F) Char
```

Then we can use them to form `LHS` terms like this:

```
meta m1        :: LHS F Int
m2 -$ v1 $- v2 :: LHS F Bool
```

4.1 Object-Language Variables and Binders

The following type class is for object-language variables and binders:

```
class Bind r where
  var :: Var r a → r a
  lam :: (Var r a → r b) → r (a → b)
```

The function `var` constructs a variable, and `lam` makes a λ-abstraction from a Haskell function. For example, the term $\lambda x.\ x + 2$ is represented as follows:

```
lam (λx → var x + 2)
```

Note that the only way to construct a value of the abstract type `Var` is using `lam`. This ensures that `Var` faithfully represents object-language variables.

The concrete representation of object-language variables uses `VAR` which is a typed newtype wrapper around a name:

```
type instance Var (LHS f) = VAR
type instance Var (RHS f) = VAR

newtype VAR a = Var Name deriving Functor
```

`VAR` has the same double role as `META` in Sect. 2.1: it is both used to identify object-language variables and as a functor that represents a variable node in a term.

The above `Var` instances both have `VAR` on the right hand side, but in Sect. 5 we will see an instance with a different right hand side.

The library uses a first-order term representation internally, despite the fact that `lam` has a higher-order type. This is possible due Axelsson and Claessen's technique for generating first-order terms from a higher-order interface [2].

4.2 Rewriting

The functions that perform higher-order rewriting have slightly different types compared to those from Sect. 2.2. One difference is that the result of rewriting is a term where each node is annotated with its set of free variables. As discussed in Sect. 3.2, we need to cache the set of free variables in order to make matching efficient.

The function `applyFirst` now has the following type:

```
applyFirst :: (..., g ∼ (f :&: Set Name))
           ⇒ (Term g → Term g → Term g)
           → [Rule (LHS f) (RHS f)]
           → Term g → Term g
```

`Term (f :&: Set Name)` is a term where each node is annotated with a set of names. The first argument to `applyFirst` is an application operator which is used when replacing applied meta-variables on the RHS of a rule. Taking this operator as an argument allows the user to choose whether to construct a redex or to reduce it right away.

In order to shield the user from the free-variable annotations, we provide the following function that turns a rewriter for annotated terms into one for non-annotated terms:

```
rewriteWith :: (..., g ∼ (f :&: Set Name))
            ⇒ (Term g → Term g) → Term f → Term f
```

A typical use of this function is

```
rewriteWith (bottomUp (applyFirst ...)) :: (...) ⇒ Term f → Term f
```

where `f` is a functor without annotation.

4.3 Quantifying over Meta-Variables

Functions such as `applyFirst` take a list of rules as argument. But most rules are of the form of Haskell functions that take extra arguments corresponding to the meta-variables used. For example, the type of `rule_add` from Sect. 2 is

```
rule_add :: ( MetaVar lhs, MetaVar rhs, Num (lhs a)
            , MetaRep lhs ∼ MetaRep rhs
            )
         ⇒ MetaRep lhs a → Rule lhs rhs
```

The `Quantifiable` type class automates the task of providing fresh meta-variables to functions like `rule_add`:

```
class Quantifiable rule where
  type RuleType rule
  quantify' :: Name → rule → RuleType rule

quantify :: (Quantifiable rule, RuleType rule ∼ Rule lhs rhs)
         ⇒ rule → Rule lhs rhs
quantify = quantify' 0

instance Quantifiable (Rule lhs rhs) where
  type RuleType (Rule lhs rhs) = Rule lhs rhs
  quantify' _ = id

instance (Quantifiable rule, m ∼ MetaId a) ⇒ Quantifiable (m → rule) where
  type RuleType (m → rule) = RuleType rule
  quantify' i rule = quantify' (i+1) (rule (MetaId i))
```

The first instance is for rules that do not have any meta-variables to quantify over. The second instance recursively quantifies one meta-variable at a time. `MetaId` is the concrete representation of meta-variables.

Using `quantify`, we can package our rules in a list of the type expected by `applyFirst`:

```
rules = [ quantify (rule_add :: _ Int → _)
        , quantify (rule_sub :: _ Int → _)
        , quantify rule_mul ]
```

Note the use of a partial type signature to constrain the type of the meta-variable, which would otherwise be ambiguous.

5 Case Study – Feldspar

The repository contains an example file[5] inspired by Feldspar that makes use of the library. In this section, we will highlight the important parts of that implementation. The interested reader is encouraged to learn more by looking at the source code.

Feldspar's expression type `Data` is defined as a newtype wrapper around a `Term` over the functor `Feld`:

```
type Feld = VAR :+: LAM :+: APP :+: NUM :+: LOGIC :+: FORLOOP

newtype Data a = Data { unData :: Term Feld }
```

`Feld` is a sum of several smaller functors, where `VAR`, `LAM` and `APP` represent the constructs of the lambda calculus, `NUM` and `LOGIC` represent numeric and logic operations, and `FORLOOP` is the Feldspar-specific for-loop.

Object-language variables are represented just as Feldspar expressions, which avoids the need to use the `var` function to introduce object-language variables:

```
type instance Var Data = Data
```

[5] https://github.com/emilaxelsson/ho-rewriting/blob/0.2/examples/Feldspar.hs.

Since we want to be able to construct for-loops in rules as well as in ordinary Feldspar expressions, we overload the for-loop using a type class:

```
class ForLoop r where
  forLoop_ :: r Int → r s → r (Int → s → s) → r s
```

The third argument to `forLoop_` has function type, so it needs to be constructed by `lam`. The following higher-order function takes care of wrapping the body in `lam`:

```
forLoop :: (ForLoop r, Bind r)
        ⇒ r Int → r s → (Var r Int → Var r s → r s) → r s
forLoop len init body = forLoop_ len init (lam $ λi → lam $ λs → body i s)
```

Exposing functions like `forLoop` to the user instead of `lam` and `forLoop_` ensures that λ-abstractions are only used at specific places. This solves the problem of matching lambdas that was mentioned in Sect. 3.2. Although restricting the use of `lam` may not be desired in general, it works well in a language like Feldspar which is essentially a first-order language with a few predefined higher-order symbols such as `FORLOOP`.

Using `forLoop`, we can now express the three for-loop rules from Sect. 1.1:

```
rule_for1 init = forLoop 0  (meta init) (λi s → _)     ⟹  meta init
rule_for2 init = forLoop _  (meta init) (λi s → var s) ⟹  meta init
rule_for3 len init body =
    forLoop (meta len) (meta init) (λi s → body -$- i)
      ⟹
    cond (meta len === 0) (meta init) (body -$- (meta len - 1))
```

The `===` operator is equality in this toy version of Feldspar.

A Feldspar simplifier is obtained by applying the simplification rules bottom-up as follows:

```
simplify :: Data a → Data a
simplify = Data . rewriteWith (bottomUp (applyFirst app rulesFeld)) . unData
```

The application operator `app` tells `applyFirst` to keep any redexes created by rewriting, and `rulesFeld` is a list of all the rules defined in this paper.

We have used `forLoop` to define rules, but we can also use it to write Feldspar expressions. Here is an example containing two for-loops that can be simplified:

```
forExample :: Data Int → Data Int
forExample a = forLoop a a (λi s → (i-i)+s) + forLoop a a (λi s → i*i+100)
```

We simplify the expression by running

```
*Main> unData $ simplify $ lam forExample
(Lam 2 (Add (Var 2) (Cond (Equal (Var 2) (Num 0)) (Var 2) (App (Lam 1 (Add
(Mul (Var 1) (Var 1)) (Num 100))) (Sub (Var 2) (Num 1))))))
```

We see that the simplifier removes both loops: the first one because it never changes the state, and the second one because its body does not refer to the previous state.

6 Related Work

Function Patterns. One of the problems solved in this paper is being able to use the same syntax both for pattern matching and construction and to hide the underlying representation of expressions. A more general solution to this particular problem is *function patterns* [1,8,17], which allow ordinary functions to be used inside patterns. When matching a term `t` against a function pattern, say `f p`, the inverse of `f` is used to compute a value to match against the argument `p`. Here, `f` can be a smart constructor whose purpose is to hide the representation of terms, or to give a typed interface using phantom types.

The recent `PatternSynonyms` extension in GHC allows the declaration of bidirectional patterns that can be used both for matching and construction. These can be seen as a restricted form of function patterns.

There may seem to be a similarity between function patterns and patterns with applied meta-variables in our library. In both cases we have patterns involving applied functions. However, there are important differences: First of all, function patterns allow *ordinary functions* inside patterns, while our library is only concerned with functions in some *object language*. Moreover, function patterns only allow using *existing* functions inside patterns; they cannot be used to synthesize function definitions the way that higher-order matching does.

Mohnen introduced *context patterns* for Haskell [13]. These are more similar to higher-order matching in that they allow meta-variables of function type (so matching can actually synthesize function definitions). However, a main difference to our work (again) is that context patterns involve actual Haskell functions rather than object-level functions.

Rewriting Libraries. Yokoyama et al. have made a library based on Template Haskell for higher-order rewriting of Haskell code [23]. Just like in our library, they restrict matching in the interest of efficiency. However, their restrictions are different: for example, meta-variables can be applied to at most one argument, but this argument can be an arbitrary pattern. It remains to be investigated whether their restrictions are suitable for the kind of syntactic rewrites that we are interested in.

There has also been work on generic, first-order rewriting libraries for Haskell [5,9,15]. In particular, the work of van Noort et al. [15] has similarities to our implementation: it uses an intensional representation of rules as data, and it generically extends data type representations with a constructor for meta-variables. The main differences are that their library works for any regular data type and that it does not support higher-order rewriting. The library by Felgenhauer, et al. [9] also has an intensional representation of rules, but uses a simpler term representation: a rose tree extended with meta-variables.

As a concrete example, here is the rule for addition with zero, expressed with our library and with the library by van Noort et al.:

```
rule_add x = 0 + meta x     ⟹  meta x  -- our library
rule_add x = Add (Num 0) x  ⟼  x       -- van Noort's library
```

`Add` and `Num` are constructors of the regular data type for expressions. It seems plausible that their library can be combined with smart constructors to get rules that look more like in our library (with the added benefit of type-safety, etc.). We see no direct reason why one could not also extend their library to higher-order rewriting, given a suitable generic representation of variables and binders.

Strategic rewriting libraries such as KURE [19] provide a rich set of strategies for building complex traversals of data. The main focus in this paper is on rewrite rules rather than strategies – `bottomUp` and `topDown` being the two main strategies presented. It would be interesting to extend our library with more strategies, or perhaps even combine it with an existing library for strategic rewriting.

Pattern Unification. Higher-order pattern unification is at the core of systems that manipulate higher-order data like Twelf [18] and λProlog [14] or type reconstruction algorithms for dependently typed languages such as Agda [16]. In such cases the unification problems that fall into the pattern fragment are solved immediately, while the others are suspended in hope that they will become tractable later when more meta-variables have been solved.

7 Discussion and Future Work

The motivation behind the presented library is for it to be used in the implementation of Feldspar. However, Feldspar is currently based on a different term representation. Our future plan is to rewrite Feldspar so that it can use the rewriting library for optimizations.

A key aspect of the library is the algorithmic efficiency of higher-order rewriting: the complexity of rule application is bounded only by the size of the rule, just like for first-order rewriting. However, we have not yet tested how well the library performs in practice for large problems. This remains as future work.

The presented library makes it possible to express higher-order rewrite rules in a safe way using clean syntax. However, one disadvantage of the library is that the error messages can be quite confusing due to the heavy type-level machinery involved. This is a common problem of embedded DSLs, but it may be solvable by recent work on type error diagnosis [20].

When writing rules like `rule_add` below, the intention is that `lhs` and `rhs` should be abstract in order to disallow inspection of the meta-variable argument.

```
rule_add :: ( ... ) ⇒ MetaRep lhs a → Rule lhs rhs
```

But rewriting functions such as `applyFirst` accept rules with the concrete representations `LHS` and `RHS`. In order to make the library safer, we should require the arguments to rewriting functions to be polymorphic in their representations. We have not yet been able to make such a solution work together with `quantify` and constraints such as `Num (lhs a)`, but we are hopeful that it can be done.

Acknowledgements. This research was funded by the Swedish Foundation for Strategic Research (in the RAWFP project). The anonymous referees provided useful input.

References

1. Antoy, S., Hanus, M.: Declarative programming with function patterns. In: Hill, P.M. (ed.) LOPSTR 2005. LNCS, vol. 3901, pp. 6–22. Springer, Heidelberg (2006)
2. Axelsson, E., Claessen, K.: Using circular programs for higher-order syntax: functional pearl. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, pp. 257–262. ACM, New York (2013)
3. Axelsson, E., Claessen, K., Dévai, G., Horváth, Z., Keijzer, K., Lyckegård, B., Persson, A., Sheeran, M., Svenningsson, J., Vajda, A.: Feldspar: a domain specific language for digital signal processing algorithms. In: 8th IEEE/ACM International Conference on Formal Methods and Models for Codesign, pp. 169–178. IEEE (2010)
4. Bahr, P., Hvitved, T.: Compositional data types. In: Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming, pp. 83–94. ACM, New York (2011)
5. Brown, N.C., Sampson, A.T.: Matching and modifying with generics. In: Draft Proceedings of Trends in Functional Programming, pp. 304–318 (2008)
6. Carette, J., Kiselyov, O., Shan, C.C.: Finally tagless, partially evaluated: tagless staged interpreters for simpler typed languages. J. Funct. Programm. **19**(5), 509–543 (2009)
7. Cervesato, I., Pfenning, F.: A linear spine calculus. J. Logic Comput. **13**(5), 639–688 (2003)
8. Dévai, G.: Extended pattern matching for embedded languages. Annales Univ. Sci. Budapestiensis de Rolando Eötvös Nominatae, Sectio Comutatorica **36**, 277–297 (2012)
9. Felgenhauer, B., Avanzini, M., Sternagel, C.: A Haskell library for term rewriting. CoRR abs/1307.2328 (2013). http://arxiv.org/abs/1307.2328
10. Goldfarb, W.D.: The undecidability of the second-order unification problem. Theoret. Comput. Sci. **13**(2), 225–230 (1981)
11. Huet, G.: Résolution d'équations dans les langages d'ordre 1, 2,.., ω. Ph.D. thesis, Université Paris VII (1976)
12. Miller, D.: A logic programming language with lambda-abstraction, function variables, and simple unification. J. logic Comput. **1**(4), 497–536 (1991)
13. Mohnen, M.: Context patterns in Haskell. In: Kluge, W. (ed.) IFL'96. LNCS, vol. 1268, pp. 41–57. Springer, Berlin, Heidelberg (1997)
14. Nadathur, G., Miller, D.: An overview of Lambda-Prolog. Technical report MS-CIS-88-40, University of Pennsylvania, Department of Computer and Information Science (1988)
15. van Noort, T., Yakushev, A.R., Holdermans, S., Jeuring, J., Heeren, B., Magalhães, J.P.: A lightweight approach to datatype-generic rewriting. J. Funct. Programm. **20**, 375–413 (2010)
16. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007
17. Oosterhof, N., Hölzenspies, P., Kuper, J.: Application patterns. In: van Eekelen, M. (ed.) Trends in Functional Programming, pp. 370–382. Tartu University Press, Tallinn (2005)
18. Pfenning, F., Schürmann, C.: System description: twelf - a meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999)

19. Sculthorpe, N., Frisby, N., Gill, A.: The kansas university rewrite engine. J. Funct. Programm. **24**, 434–473 (2014)
20. Serrano, A., Hage, J.: Feedback upon feedback. Presented at Trends in Functional Programming (2015). ftp://ftp-sop.inria.fr/indes/TFP15/TFP2015_submission_22.pdf
21. Swierstra, W.: Data types à la carte. J. Funct. Programm. **18**, 423–436 (2008)
22. Wierzbicki, T.M.: Complexity of the higher order matching. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 82–96. Springer, Heidelberg (1999)
23. Yokoyama, T., Hu, Z., Takeichi, M.: Design and implementation of deterministic higher-order patterns (2005). http://takeichi.ipl-lab.org/yicho/YoHT05.pdf

Towards a Theory of Reach

Jonathan Fowler(✉) and Graham Huttom

School of Computer Science, University of Nottingham, Nottingham, UK
psxjf@exmail.nottingham.ac.uk

Abstract. When testing a program, there are usually some parts that are rarely executed and hence more difficult to test. Finding inputs that guarantee that such parts are executed is an example of a *reach problem*, which in general seeks to ensure that targeted parts of a program are always executed. In previous work, Naylor and Runciman have developed a reachability solver for Haskell, based on the use of lazy narrowing from functional logic programming. Their work was focused on practical issues concerning implementation and performance. In this paper, we lay the groundwork for an underlying theory of such a system, by formally establishing the correctness of a simple reach solver.

1 Introduction

A desirable goal of software testing is for every reachable expression within a program to contribute to at least one test execution of the program. The testing then exhibits program coverage. Random property testing systems such as Quickcheck [3] often cover most of a program, but particularly hard to reach expressions may remain untested. The *Reach* system [12] was developed to address this problem, by generating inputs that execute a particular target expression. By using the Haskell Program Coverage (HPC) tool [5] to find expressions which are not tested by Quickcheck, and Reach to generate inputs that execute these expressions, the goal of program coverage can be achieved.

Work to date on the Reach system by Naylor and Runciman [12,13] has focused on the implementation and performance of various underlying solvers. In this paper, we investigate a formal definition for the *reach problem*, and how the forward solver defined in their original paper [12] can be shown to be correct. Having such a theory is important to check the correctness of more complex solvers, such as backwards solver described in Naylor's thesis [13]. The act of formalisation also opens up new potential avenues for further research into alternate evaluation strategies, as discussed in Sect. 10.

The forward reach solver developed by Naylor and Runciman uses a *lazy narrowing* evaluation strategy adapted from functional logic programming. Lazy narrowing can be thought of as an extension to the semantics of a non-strict language to include reduction rules for free variables. The basic idea is that when the value of a free variable is required for a case analysis to proceed, we bind the free variable to each possible alternative form that it may have. To focus on the essence of the problem, we initially consider a minimal language

M. Serrano and J. Hage (Eds.): TFP 2015, LNCS 9547, pp. 22–39, 2016.
DOI: 10.1007/978-3-319-39110-6_2

(Sect. 3) that includes only Peano-encoded natural numbers, a target expression, and case expressions. Abstracting away from the details of a real language such as Haskell we keep the presentation neat and concise but still include enough detail to express and understand the properties of the reach problem and lazy narrowing. Within the context of this minimal language we:

- Extend the language with free variables, and give a precise definition for the 'reach problem' in this setting (Sect. 4);
- Define a lazy narrowing semantics for the extended language and use the semantics to define a forward reach solver (Sect. 5);
- Show that the lazy narrowing semantics is sound and complete with respect to the original semantics, and that our reach solver is correct (Sect. 6);
- Provide a mechanical verification of our results in the Agda system, and make the proof scripts freely available online (Sect. 7);
- Describe how the language can be extended with a number of additional features, and extend the Agda formalisation accordingly (Sect. 8).

We present proofs for our main results based on a number of lemmas, but for brevity do not provide proofs for the lemmas and refer the interested reader to the accompanying Agda code for the details [4]. The intended audience for the article is functional programmers with a basic knowledge of semantics. No prior knowledge of Reach is assumed; an introduction is given in Sect. 2.

2 The Reach Problem

Reach [12] is a tool for Haskell that can be used help achieve program coverage. A *reach problem* is a Haskell program with a marked target expression and source function. The goal is to find an input to the source function that entails evaluation of the target expression. The target is typically placed in a rarely evaluated expression within the program. The inputs generated from the running of the Reach solver can then be used as test cases for these expressions.

As an example, consider a simplified version of a *balance* function from the standard library *Data.Map*. The *balance* function takes a binary tree and redistributes the tree when one sub-tree contains substantially more elements than the other, in this case four times as many:

$$
\begin{aligned}
&\mathit{balance} :: \mathit{Tree}\ a \to \mathit{Tree}\ a \\
&\mathit{balance}\ (\mathit{Leaf}\ a) = \mathit{Leaf}\ a \\
&\mathit{balance}\ (\mathit{Node}\ lt\ rt) \\
&\quad \mid \mathit{size}\ rt \geqslant 4 * \mathit{size}\ lt = \mathit{balanceToL}\ lt\ rt \\
&\quad \mid \mathit{size}\ lt \geqslant 4 * \mathit{size}\ rt = \mathit{balanceToR}\ lt\ rt \\
&\quad \mid \mathit{otherwise} \qquad\qquad = \mathit{Node}\ lt\ rt
\end{aligned}
$$

When testing this function randomly, for example using a *standard generator* for a Quickcheck property [9], the case when the tree is already balanced according to the above definition is tested far more often than the interesting case

when the tree needs balancing. By replacing the branch of the guard requiring the tree to be right-heavy with a target expression, indicated by •, we create a Reach problem which will generate input trees that require balancing.

$$
\begin{aligned}
&\mathit{balance} :: \mathit{Tree}\ a \rightarrow \mathit{Tree}\ a \\
&\mathit{balance}\ (\mathit{Leaf}\ a) = \mathit{Leaf}\ a \\
&\mathit{balance}\ (\mathit{Node}\ \mathit{lt}\ \mathit{rt}) \\
&\quad \mid \mathit{size}\ \mathit{rt} \geqslant 4 * \mathit{size}\ \mathit{lt} = \bullet \\
&\quad \mid \mathit{size}\ \mathit{lt} \geqslant 4 * \mathit{size}\ \mathit{rt} = \mathit{balanceToR}\ \mathit{lt}\ \mathit{rt} \\
&\quad \mid \mathit{otherwise} \qquad\quad = \mathit{Node}\ \mathit{lt}\ \mathit{rt}
\end{aligned}
$$

A solution to the Reach problem with *balance* as the input function is a tree which satisfies the first guard, such as the following:

$$\mathit{Node}\ (\mathit{Leaf}\ 0)\ (\mathit{Node}\ (\mathit{Node}\ (\mathit{Leaf}\ 1)\ (\mathit{Leaf}\ 2))\ (\mathit{Node}\ (\mathit{Leaf}\ 5)\ (\mathit{Leaf}\ 2)))$$

This tree can then be used as an input to the original *balance* function to ensure that the auxiliary function *balanceToL* is executed as part of testing. In a similar manner, we can move the target expression to the second branch of the guard to find a tree which ensures that *balanceToR* is executed.

2.1 Forward Reach

In this section we introduce the primary reach solver, Forward Reach, defined by Naylor and Runciman [12,13]. Forward Reach uses *lazy narrowing* in order to generate inputs efficiently. Lazy narrowing is a concept from functional logic programming [2,7] and can be described as the natural extension of a non-strict semantics to a language with free variables. Free variables are only bound when their value is required for evaluation to proceed.

To illustrate, we give an example of a lazy narrowing reach solver in action. We show the first steps of an analysis of the *balance* function from the previous section. Each state during evaluation is given by an expression and a *substitution*, a mapping which is an accumulation of the free variable bindings up to the current point of evaluation. For our example, the initial expression is *balance* x and the initial substitution is the trivial mapping $x \mapsto x$ from x to itself.

1) $\{x \mapsto x\}$
   balance x

Starting with the trivial mapping rather than the traditional empty mapping helps with the formalisation, as discussed further in Sect. 4.1. The first step of evaluation is to inline the definition for *balance* x:

2) $\{x \mapsto x\}$
   case x **of**
   $\quad \mathit{Leaf}\ a \rightarrow \mathit{Leaf}\ a$
   $\quad \mathit{Node}\ \mathit{lt}\ \mathit{rt} \rightarrow \ldots$

In order for evaluation to continue the value of the free variable x is now required, which necessitates a *narrowing* step. To begin with, the variable is bound to the leaf constructor for trees by refining the substitution to $x \mapsto \mathit{Leaf}\ x'$, and updating the expression being evaluated accordingly.

3) $\{x \mapsto \mathit{Leaf}\ x'\}$
case $\mathit{Leaf}\ x'$ **of**
$\mathit{Leaf}\ a \rightarrow \mathit{Leaf}\ a$
$\mathit{Node}\ lt\ rt \rightarrow \ldots$

Note that the introduction of a new variable x' is not strictly necessary above, as the manner in which substitutions are later formalised ensures that the variables on the left and right sides of a substitution are independent. Hence, we could equally well use the substitution $x \mapsto \mathit{Leaf}\ x$ above, and indeed this simpler approach – which avoids the need to generate a fresh variable name – is used in our subsequent formalisation in Sect. 5. Now that the form of the expression is known, we can reduce the case expression itself:

4) $\{x \mapsto \mathit{Leaf}\ x'\}$
$\mathit{Leaf}\ x'$

Evaluation of this execution path terminates with the value $\mathit{Leaf}\ x'$. In this case, the target has not been evaluated so the input $\mathit{Leaf}\ x'$ is not a solution to the reach problem, independent of any value substituted for x'. Evaluation now backtracks and x is bound to the node constructor for trees. After the narrowing step and following reduction of the case expression we have:

5) $\{x \mapsto \mathit{Node}\ x_\mathbf{l}\ x_\mathbf{r}\}$
if $\mathit{size}\ x_\mathbf{l} \geqslant 4 * \mathit{size}\ x_\mathbf{r}$ **then** $\bullet$ **else** ...

Analysis will continue with evaluation of the expression $\mathit{size}\ x_\mathbf{l} \geqslant 4 * \mathit{size}\ x_\mathbf{r}$. Inputs that evaluate to the target will be collected and evaluation will continue until a set number of solutions is found or a given termination condition is reached, e.g. the input has been enumerated to a particular depth.

Lazy narrowing has two key efficiency benefits over the naive approach in which possible inputs are enumerated and evaluated from the beginning each time. First of all, and most importantly, it allows for portions of the input domain to be discarded or accepted if the evaluation concludes while there are still free variables in the substitution, as the same conclusion can be drawn for any input formed by replacing these free variables. This can greatly reduce the search space. For example, above we were able to discard any input of the form $\mathit{Leaf}\ x'$. Secondly, some evaluation is shared between different inputs if they have common structure. In particular, their evaluation is shared up to the point where their differences cause execution to take separate branches.

3 A Minimal Language

In this section we introduce the minimal language that we will use for the rest of paper. The language is not suitable for actual programming, but does provide

enough structure to describe the key mechanisms of lazy narrowing. To this end the language has only one type, Peano natural numbers, which provides the simplest example type for showing the recursive mechanics of narrowing. The grammar for expressions of the language is defined as follows:

$$
\begin{aligned}
\mathit{Exp} ::= \;& \texttt{Zero} \\
\mid \;& \texttt{Suc}\ \mathit{Exp} \\
\mid \;& \bullet \\
\mid \;& \textbf{case}\ \mathit{Exp}\ \textbf{of}\ \mathit{Exp}\ \mathit{Alt} \\
\mid \;& \texttt{var}\ \mathit{Var} \\
\mathit{Alt} ::= \;& \texttt{Suc}\ \mathit{Var} \to \mathit{Exp} \\
\mathit{Val} ::= \;& \texttt{Zero} \mid \texttt{Suc}\ \mathit{Val}
\end{aligned}
$$

That is, an expression is either a natural number, a target expression $\bullet$, a case expression, or a variable from some given set *Var* of names. Case expressions have the form $\textbf{case}\ e\ \textbf{of}\ e_0\ f$, where the first alternative is the `Zero` branch and the second alternative is the `Suc` branch, which can depend on its argument variable. Expressions are assumed to be closed; variables only appear within the case expression in which they are bound. The values of the language are simply the natural numbers. We do not regard the target expression itself as a value, because our intended interpretation is that the values are 'normal' results.

Note that the language does not contain functions or recursion, as these are not required to study the 'essence' of lazy narrowing. We do however provide an additional Agda formalisation that incorporates these features, as discussed in Sect. 8. One might also ask why the target expression, which is specific to the Reach problem, is already included in the above language. The reason is simply for convenience: if the target expression was excluded we would need to extend both the syntax and semantics when we later define the Reach problem, whereas including it here means that we only need to extend the syntax.

The behaviour of expressions is defined as a small-step operational semantics, $\to \subseteq \mathit{Exp} \times \mathit{Exp}$, by means of the following inference rules:

$$
\frac{}{\textbf{case}\ \bullet\ \textbf{of}\ e_0\ f \to \bullet}\ \textsc{target} \qquad \frac{}{\textbf{case}\ \texttt{Zero}\ \textbf{of}\ e_0\ f \to e_0}\ \textsc{case-z}
$$

$$
\frac{}{\textbf{case}\ \texttt{Suc}\ e\ \textbf{of}\ e_0\ (\texttt{Suc}\ v \to e') \to e'[v := e]}\ \textsc{case-suc}
$$

$$
\frac{e \to e'}{\textbf{case}\ e\ \textbf{of}\ e_0\ f \to \textbf{case}\ e'\ \textbf{of}\ e_0\ f}\ \textsc{subj}
$$

Using a small-step semantics enforces a clear order of evaluation, and supports a natural extension to lazy narrowing. If the case subject is a `Zero` or `Suc` then the semantics are standard, where $e'[v := e]$ denotes the substitution of variable v by the expression e in the expression e' in a capture avoiding manner. The

target expression behaves in the same way as an error value, i.e. it is always propagates through a case expression to the top level, on the basis that once we have found a target no further evaluation is required.

When applying the semantics in practice, we often use the reflexive transitive closure, $\rightarrow^*$, which is defined in the normal manner:

$$\frac{e \rightarrow e' \qquad e' \rightarrow^* e''}{e \rightarrow^* e''}\ \text{SEQ} \qquad\qquad \frac{}{e \rightarrow^* e}\ \text{REFL}$$

The semantics can be shown by standard methods to be normalising (always terminates in a finite number of steps) and deterministic (always produces a single possible result). However, neither property is a requirement for the definition of the Reach problem or the correctness result which follows.

4 Adding Free Variables

To specify the Reach problem we require a notion of *free* variables. One possibility is to simply allow our expressions to be open, letting the existing variables be free. Although this is the approach taken in the original Reach work [12,13], we choose to syntactically separate the free variables as an extension of the language. Our reason for making this choice is that free variables are independent of the normal variables of a language; for example, it is easy to make a similar extension to a language that does not have any form of variables.

The extended grammar for expressions is defined below, in which each rule is now parameterised by a set X of free variables, and expressions and values are extended with free variables of the form `fvar` X. Note that we do not require the set of variables for an expression to be minimal, i.e. the set may contain variables that are not used in the expression.

$$\begin{aligned}
Exp_X \ ::= \ & \texttt{Zero} \\
\mid \ & \texttt{Suc}\ Exp_X \\
\mid \ & \bullet \\
\mid \ & \textbf{case}\ Exp_X\ \textbf{of}\ Exp_X\ Alt_X \\
\mid \ & \texttt{var}\ Var \\
\mid \ & \texttt{fvar}\ X \\
Alt_X \ ::= \ & \texttt{Suc}\ Var \rightarrow Exp_X \\
Val_X \ ::= \ & \texttt{Zero} \mid \texttt{Suc}\ Val_X \mid \texttt{fvar}\ X
\end{aligned}$$

We will view values of type Val_X as *partial values*, in the sense that they may contain undefined components represented by the free variables. We can also view the original grammars as special cases of the free variable versions in which the free variable sets are empty, i.e. $Exp \equiv Exp_\emptyset$, $Alt \equiv Alt_\emptyset$ and $Val \equiv Val_\emptyset$.

4.1 Substitutions

An input to an expression is a mapping from its free variables to values. In order to define this formally, we first make a slight detour to introduce the more

general notation of a substitution, which will be used later in lazy narrowing. A *substitution* of type $X \to Y$ is a mapping from the set of free variables X to partial values that contain free variables from the set Y:

$$Sub_{X \to Y} \;=\; X \to Val_Y$$

Defining substitutions in this manner rather than as a partial mapping from an infinite set of variables results in a simpler formalisation in Agda. In particular, incorporating the set of variables for the domain and range directly into the type removes the need to add the variable sets as constraints later on. A second benefit of this approach is that it yields a monadic interpretation to the composition of substitutions. Given this representation the traditional empty map becomes the trivial map in which each variable is mapped to itself.

Using our notion of substitution, an *input* to an expression can then be viewed as a special case when the set of free variables in the result is empty:

$$Inp_X \;=\; Sub_{X \to \emptyset}$$

We denote substitutions by σ and inputs by τ. The process of applying a substitution is defined recursively in the normal way:

$$\begin{array}{ll}
_\,[_] & :: \; Exp_X \to Sub_{X \to Y} \to Exp_Y \\
\texttt{Zero}\;[\sigma] & = \texttt{Zero} \\
\texttt{Suc}\;\; e\;[\sigma] & = \texttt{Suc}\;(e\;[\sigma]) \\
\bullet\;[\sigma] & = \bullet \\
\textbf{case}\; e\; \textbf{of}\; e_0\;(\texttt{Suc}\;\; v \to e')\;[\sigma] & = \textbf{case}\; e\;[\sigma]\; \textbf{of}\; e_0\;[\sigma]\;(\texttt{Suc}\;\; v \to e'\;[\sigma]) \\
\texttt{var}\;\; v\;[\sigma] & = \texttt{var}\;\; v \\
\texttt{fvar}\;\; x\;[\sigma] & = \sigma\; x
\end{array}$$

4.2 Reachability

We can now specify the meaning of *reachability* within our framework. Given an expression $e \in Exp_X$ with free variables X, the set of inputs $reach(e) \subseteq Inp_X$ that reach the target expression is defined as follows:

$$\tau \in reach(e) \quad \Longleftrightarrow \quad e[\tau] \to^* \bullet$$

That is, an input τ that provides values for the free variables in expression e satisfies the reachability condition *iff* the input applied to the expression evaluates to the target. This equivalence describes what it means for a given input to reach the target, but does not describe a specific reach problem. An example for such problem might be to find a specific input that satisfies reachability, or to show that none exists. In most languages, but not in our minimal language, the problem is undecidable and therefore an additional termination criterion is included, e.g. find a solution up to a given search depth.

A naive approach to implementing a reach solver is to search for a solution by brute force enumeration and evaluation of all possible inputs. Clearly, however, this is not very efficient. Instead, Naylor and Runciman [12] implement an approach based on lazy narrowing which proves far more efficient. This approach shares evaluation, where possible, across the input domain.

5 Lazy Narrowing Semantics

In this section we define a semantics for our minimal language extended with free variables, based upon the notion of *lazy narrowing*, a symbolic evaluation strategy from functional logic programming. As illustrated in Sect. 2, the basic idea of lazy narrowing is that when evaluation of an expression is suspended on the value of a free variable, we allow evaluation to proceed by performing a *narrowing step*, in which each partial value that the variable could have is considered in turn. As evaluation proceeds a substitution is gradually built up which tracks the instantiation of free variables.

5.1 Preliminaries

We begin by defining a number of concepts that are used in our formalisation of the notion of lazy narrowing, in the form of suspended expressions, minimal narrowing sets, and the composition of substitutions.

Suspended Expressions. An expression e is *suspended* on a free variable x, denoted by $e \multimap x$, if the value of the variable is required for evaluation of the expression to proceed any further. For our language, the relation $\multimap \subseteq Exp_X \times X$ can be defined by the following two inference rules:

$$\frac{}{\texttt{fvar}\, x \multimap x}\ \text{SUSP} \qquad \frac{e \multimap x}{\textbf{case}\ e\ \textbf{of}\ e_0\ f \multimap x}\ \text{SUBJ-SUSP}$$

That is, free variables are themselves suspended, and a case expression is suspended if its subject expression is suspended. Expressions that are suspended can make no further transitions in our small-step operational semantics from Sect. 3. However, the converse is not true. In particular, values and the target expression cannot make further transitions, but are not suspended.

Minimal Narrowing Set. When an expression is suspended there is a set of possible narrowing steps that can be performed. However, in order to maximise laziness, each of the steps that are considered should be *minimal*, in the sense that it should only instantiate the free variable just enough to allow evaluation to continue, and no further. For our language, in which the only values are natural numbers, this means replacing a free variable x by either `Zero` or `Suc` (`fvar` x), the two possible forms that a natural number can have.

To formalise this idea, we begin by writing $x \,/\, a$ for the one-point substitution that maps the free variable $x \in X$ to the partial value $a \in Val_Y$ and leaves all other variables in X unchanged, defined as follows:

$$\begin{aligned}
&(/) && :: (x \in X) \to Val_Y \to Sub_{X \to X[x/Y]} \\
&(x \,/\, a)\ x' \mid x \equiv x' && = a \\
&\qquad\quad \mid otherwise && = \texttt{fvar}\ x'
\end{aligned}$$

The return type of the substitution is given by $X\ [x\ /\ Y] = (X - \{x\}) \cup Y$, in which the element $x \in X$ is replaced by the set Y. Note that the type of $(/)$ depends on the name of the variable x, i.e. the operator has a *dependent* type. Being precise in this manner helps to simplify our Agda formalisation. Using this operator we can now define the *minimal narrowing set* $Narr_X(x)$ of a free variable $x \in X$ by replacing x by the two possible forms that it may have:

$$Narr_X(x) \ = \ \{x/\,\texttt{Zero},\ \ x/\,\texttt{Suc}\ (\texttt{fvar}\ x)\}$$

This set has two properties that play an important role in *completeness* of the lazy narrowing semantics. Firstly, the minimal narrowing set itself obeys a notion of completeness, in the sense that for every input that is possible before the narrowing there exists a substitution in which the input remains possible. And secondly, each substitution in the minimal narrowing set is *advancing*, in that it always instantiates a variable. These properties are formalised in Sect. 6.2.

Composition of Substitutions. As evaluation proceeds under lazy narrowing, we will construct a substitution in a compositional manner from smaller components. In order to define a composition operator for substitutions, we first note that *Val* forms a monad under the following definitions:

$$\begin{array}{ll}
\mathit{return} & :: X \to \mathit{Val}_X \\
\mathit{return} & = \texttt{fvar} \\
(\gg\!=) & :: \ \mathit{Val}_X \to (X \to \mathit{Val}_Y) \to \mathit{Val}_Y \\
\texttt{Zero} \gg\!= \sigma & = \texttt{Zero} \\
\texttt{Suc}\ e \gg\!= \sigma & = \texttt{Suc}\ (e \gg\!= \sigma) \\
\texttt{fvar}\ x \gg\!= \sigma & = \sigma\ x
\end{array}$$

We note in passing that this is the *free monad* of the underlying functor for the natural numbers. Using the $\gg\!=$ operator for this monad it is then straightforward to define the composition operator for substitutions:

$$\begin{array}{l}
(\gg\!\Rightarrow) \quad :: \ \mathit{Sub}_{X \to Y} \to \mathit{Sub}_{Y \to Z} \to \mathit{Sub}_{X \to Z} \\
\sigma \gg\!\Rightarrow \sigma' = \lambda a \to \sigma\ a \gg\!= \sigma'
\end{array}$$

Moreover, expanding out the definition of *Sub* in the type for the $\gg\!\Rightarrow$ operator gives $(X \to \mathit{Val}\ Y) \to (Y \to \mathit{Val}\ Z) \to (X \to \mathit{Val}\ Z)$, which corresponds to the standard notion of *Kleisli composition* for the *Val* monad.

Along with the monad laws we require one more law, relating the composition of substitutions to the application of a substitution.

Lemma 1. *The sequential application of substitutions to an expression is equivalent to the application of the composed substitutions to the expression:*

$$e[\sigma][\sigma'] \ \equiv \ e[\sigma \gg\!\Rightarrow \sigma']$$

5.2 Semantics

We now have all the ingredients required to define a lazy narrowing semantics for our minimal language. A step in the new semantics is either:

- a single step in the original semantics; or
- a minimal narrowing step, if the expression is suspended.

To keep track of the substitutions that are applied during narrowing, we write $e \rightsquigarrow \langle e', \sigma \rangle$ to mean that expression e can make the transition to expression e' in a single step, where σ is the substitution that has been applied in the case of a narrowing step. In the case of a step in the original semantics, we simply return the identity substitution, which is given by the *return* operator of the *Val* monad. More formally, we define a transition relation $\rightsquigarrow \subseteq Exp_X \times (Exp_Y \times Sub_{X \to Y})$ for lazy narrowing by the following two inference rules:

$$\frac{e \to_X e'}{e \rightsquigarrow \langle e',\ return \rangle}\ \text{PROM} \qquad \frac{e \multimap x \qquad \sigma \in Narr_X(x)}{e \rightsquigarrow \langle e[\sigma],\ \sigma \rangle}\ \text{NARR}$$

The first rule promotes transitions from the original semantics to the new semantics, where $\to_X \subseteq Exp_X \times Exp_X$ is the trivial lifting of the transition relation $\to \subseteq Exp \times Exp$ to operate on expressions with free variables in the set X, for which the inference rules remain syntactically the same as previously except that they now operate on expressions of a more general form. The second rule applies a minimal narrowing step to a suspended expression.

The definition of how to sequence steps in our extended semantics, which takes into account the additional presence of substitutions, is given by a relation $\rightsquigarrow^+$ that is defined by the following two rules:

$$\frac{e \rightsquigarrow \langle e',\ \sigma \rangle \qquad e' \rightsquigarrow^+ \langle e'',\ \tau \rangle}{e \rightsquigarrow^+ \langle e'',\ \sigma \gg\!\!> \tau \rangle}\ \text{SEQ} \qquad \frac{e \in Exp_X \qquad \tau \in Inp_X}{e \rightsquigarrow^+ \langle e[\tau],\ \tau \rangle}\ \text{FILL}$$

The first rule simply composes the substitutions from the two component reductions. The second rule adds a final narrowing step to the end of a reduction sequence that instantiates any remaining free variables. The reason for including a final narrowing step is that it simplifies both the definition of forward reachability and its relationship to the original semantics.

5.3 Forward Reachability

Finally, we can now give an alternative characterisation of reachability using our lazy narrowing semantics. Given an expression $e \in Exp_X$, the set of inputs $reach_F(e) \in Inp_X$ that reach the target expression is defined as follows:

$$\tau \in reach_F(e) \iff e \rightsquigarrow^+ \langle \bullet,\ \tau \rangle$$

That is, an input τ satisfies the forward reachability condition *iff* there is a lazy narrowing reduction sequence that ends with the target and the given

input. The key difference with our original definition of reachablity in Sect. 4.2 is that our new semantics *constructs* an input substitution during the reduction sequence, whereas the original semantics requires that we are *given* a substitution so that it can be applied prior to starting the reduction process. In the next section we show that these two notions of reachability coincide.

6 Correctness of the Narrowing Semantics

To prove that forward reachability is equivalent to the original definition, we first formalise the relationship between the lazy narrowing semantics and the original semantics. This relationship is characterised by two properties, *soundness* and *completeness*, which are proved using a number of lemmas. The proofs of the lemmas themselves are provided in the associated Agda formalisation.

6.1 Soundness

Lemma 2. *A transition in the original semantics can be lifted through a substitution. Given a substitution* $\sigma \in Sub_{X \to Y}$*, we have:*

$$e \to_X e' \implies e[\sigma] \to_Y e'[\sigma]$$

Theorem 1 (Soundness). *For every reduction sequence in the lazy narrowing semantics there is a corresponding sequence in the original semantics:*

$$e \rightsquigarrow^+ \langle e',\ \tau \rangle \quad \implies \quad e[\tau] \to^* e'$$

Proof. The proof proceeds by rule induction on the definition for the narrowing relation $\rightsquigarrow^+$, for which there are three cases to consider.

Case 1. In the base case when the narrowing is a simple application of

$$\frac{}{e \rightsquigarrow^+ \langle e[\tau],\ \tau \rangle}\ \text{FILL}$$

the goal follows immediately from the reflexivity of $\to^*$:

$$\frac{}{e[\tau] \to^* e[\tau]}\ \text{REFL}$$

Case 2. There are two inductive cases to consider, depending on the nature of the first reduction in a narrowing sequence. We first consider the case when the reduction is a narrowing step, constructed as follows:

$$\text{SEQ}\ \frac{\text{NARR}\ \dfrac{e \multimap x \qquad \sigma \in Narr_X(x)}{e \rightsquigarrow \langle e[\sigma],\ \sigma \rangle} \qquad e[\sigma] \rightsquigarrow^+ \langle e',\ \tau \rangle}{e \rightsquigarrow^+ \langle e',\ \sigma \gg\!= \tau \rangle}$$

We are now free to use the three assumptions $e \multimap x$, $\sigma \in Narr_X(x)$ and $e[\sigma] \rightsquigarrow^+ \langle e',\ \tau \rangle$ in our proof. In this case, we only require the third of these assumptions

in order to verify our goal, by first using the induction hypothesis (IH) $e[\sigma] \rightsquigarrow^+ \langle e',\ \tau\rangle \implies e[\sigma][\tau] \to^* e'$, and then applying Lemma 1:

$$\dfrac{\dfrac{e[\sigma] \rightsquigarrow^+ \langle e',\ \tau\rangle}{e[\sigma][\tau] \to^* e'}\ \text{IH}}{e[\sigma \gg\!\!> \tau] \to^* e'}\ \text{LEMMA 1}$$

Case 3. We now consider the case when the first reduction is a promoted reduction from the original language, constructed as follows:

$$\text{PROM}\ \dfrac{\dfrac{e \to_X e'}{e \rightsquigarrow \langle e',\ return\rangle} \qquad e' \rightsquigarrow^+ \langle e'',\ \tau\rangle}{e \rightsquigarrow^+ \langle e'',\ return \gg\!\!> \tau\rangle}\ \text{SEQ}$$

In this case our goal can then be verified by lifting the reduction from the original language through the input substitution using Lemma 2, sequencing with the result of applying the induction hypothesis to the remaining reduction sequence, and finally applying an identity law for Kleisli composition:

$$\text{LEMMA 2}\ \dfrac{\dfrac{\dfrac{e \to_X e'}{e[\tau] \to e'[\tau]} \qquad \dfrac{e' \rightsquigarrow^+ \langle e'',\ \tau\rangle}{e'[\tau] \to^* e''}\ \text{IH}}{e[\tau] \to^* e''}\ \text{SEQ}}{e[return \gg\!\!> \tau] \to^* e''}\ \text{ID}$$

□

Although the above proof was presented specifically for the specific case of lazy narrowing semantics, it is not dependent on the properties of the narrowing set or the condition for applying a narrowing step. Therefore the proof is also valid for any narrowing set and any applicability condition.

6.2 Completeness

Definition 1. We exploit two pre-orderings on substitutions, which respectively capture the idea of one substitution being a *prefix* or *suffix* of another:

$$\sigma_1 \sqsubseteq \sigma_2 \iff \exists \sigma'.\ \sigma_1 \gg\!\!> \sigma' \equiv \sigma_2$$

$$\sigma_1 \leqslant \sigma_2 \iff \exists \sigma'.\ \sigma' \gg\!\!> \sigma_1 \equiv \sigma_2$$

Lemma 3. *If the source expression of a transition in the original semantics is not suspended then the transition can be 'unlifted'. Given a substitution $\sigma \in Sub_{X \to Y}$ and a transition $e[\sigma] \to_Y e'$ for which $e \not\multimap x$, we have:*

$$\exists e'_\sigma.\ e \to_X e'_\sigma\ \wedge\ e'_\sigma[\sigma] \equiv e'$$

Lemma 4. *The lazy narrowing set is complete. For every input there is a substitution in the narrowing set that is a prefix of the input:*

$$\forall x \in X, \tau \in Inp_X \,.\, \exists \sigma \in Narr_X(x).\ \sigma \sqsubseteq \tau$$

Lemma 5. *The lazy narrowing set is advancing. The identity substitution is a strict prefix of every substitution in the narrowing set:*

$$\forall x \in X,\ \sigma \in Narr_X(x).\ \ return \sqsubset \sigma$$

Theorem 2 (Completeness). *For every reduction sequence in the original semantics there is a corresponding reduction in the lazy narrowing semantics:*

$$e[\tau] \to^* e' \quad \Longrightarrow \quad e \rightsquigarrow^+ \langle e',\ \tau \rangle$$

Proof. The proof proceeds by rule induction on the definition for the evaluation relation $\to^*$, for which there are three cases to consider.

Case 1. In the base case when the evaluation is just reflexivity

$$\frac{}{e[\tau] \to^* e[\tau]}\ \text{REFL}$$

the goal follows immediately by instantiating free variables:

$$\frac{}{e \rightsquigarrow^+ \langle e[\tau],\ \tau \rangle}\ \text{FILL}$$

Case 2. There are two inductive cases to consider, depending on whether or not the expression e is suspended when the sequencing rule is applied:

$$\frac{e[\tau] \to e' \qquad e' \to^* e''}{e[\tau] \to^* e''}\ \text{SEQ}$$

In the case when e is not suspended our goal can be verified as follows, in which the two branches of the proof tree exploit the two conclusions from Lemma 3:

$$\dfrac{\dfrac{\dfrac{\text{LEMMA 3}\ \dfrac{}{e \to e'_\tau}}{\text{PROM}\ \dfrac{}{e \rightsquigarrow \langle e'_\tau,\ return \rangle}} \qquad \dfrac{\dfrac{e' \to^* e''}{e'_\tau[\tau] \to^* e''}\ \text{LEMMA 3}}{e'_\tau \rightsquigarrow^+ \langle e'',\ \tau \rangle}\ \text{IH}}{e \rightsquigarrow^+ \langle e'',\ return \gg\!\!> \tau \rangle}\ \text{SEQ}}{e \rightsquigarrow^+ \langle e'',\ \tau \rangle}\ \text{ID}$$

Case 3. Finally, when e is suspended on x, because the narrowing set $Narr(x)$ is complete (Lemma 4) there is a substitution in this set that is a prefix of the input τ, i.e. a substitution $\sigma \in Narr(x)$ and input τ' for which $\tau \equiv \sigma \gg\!\!> \tau'$. Based upon this observation our goal can then be verified as follows:

$$\dfrac{\dfrac{\dfrac{\text{NARR}\ \dfrac{e \multimap x \qquad \sigma \in Narr(x)}{e \rightsquigarrow \langle e[\sigma],\ \sigma \rangle} \qquad \dfrac{\dfrac{e[\tau] \to^* e'}{e[\sigma][\tau'] \to^* e'}\ \text{LEMMA 1}}{e[\sigma] \rightsquigarrow^+ \langle e',\ \tau' \rangle}\ \text{IH}}{e \rightsquigarrow^+ \langle e',\ \sigma \gg\!\!> \tau' \rangle}\ \text{SEQ}}{e \rightsquigarrow^+ \langle e',\ \tau \rangle}\ \text{LEMMA 4}$$

Well-foundedness. In the third case above, we need to explicitly verify that the induction is well-founded as the induction hypothesis is not trivially smaller in this case. Instead, with each iteration the *input* gets smaller. To formalise this well-foundedness neatly and generally, we restrict our notion of substitutions $Sub_{X \to Y}$ to the case when the free variable sets X and Y are finite, and every variable in Y appears in the result of the substitution. For our purposes this leads to no loss of generality and all of our definitions satisfy these restrictions. With these in place, we then have the following two results, which together with Lemma 5 ensures that the use of induction in the third case is well-founded.

Lemma 6. *The suffix relation $<$ is well-founded. For any substitution τ_0, there only exists finite chains of substitutions τ_i such that:*

$$\tau_n < \ldots < \tau_1 < \tau_0$$

Lemma 7. *A suffix formed by an advancing prefix is strict.*

$$\sigma \succ\!\!\!\gg \sigma_1 \equiv \sigma_2 \ \wedge \ return \sqsubset \sigma \implies \sigma_1 < \sigma_2 \qquad \square$$

Whereas the soundness proof was independent of the properties of the narrowing set and the condition for its applicability, the completeness proof relies on the fact that the narrowing set is complete and advancing, and that narrowing steps can always be applied when an expression is suspended.

6.3 Correctness

Using the soundness and completeness results, it is now straightforward to prove that our two notions of reachability are equivalent:

Theorem 3 (Correctness). *For all expressions $e \in Exp_X$:*

$$reach_F(e) \equiv reach(e)$$

Proof.

$$\begin{aligned} \tau \in reach_F(e) &\iff e \rightsquigarrow^+ \langle \bullet, \tau \rangle && \text{(by definition)} \\ &\iff e[\tau] \to^* \bullet && \text{(Theorems 1 and 2)} \\ &\iff \tau \in reach(e) && \text{(by definition)} \end{aligned}$$

7 Agda Formalisation

Our correctness result has also been formalised in the Agda [14]. The Agda formalisation follows the presentation given in the paper closely: the language grammar and semantic rules convert directly to inductive datatypes, and rule induction translates to recursive dependent functions. A proof of the main result and all associated lemmas is available online from:

http://tinyurl.com/reachtheory

Using Agda brings a number of important benefits. First of all, it provides a guarantee that our results are correct. Secondly, it helped guide the development of our theory and proofs, resulting in a number of simplifications. For example, when translating our original formalisation into Agda we found that it contained a subtle error. The process of correcting the error also pointed towards a neater theory. In particular, our original lazy narrowing formulation kept the substitution as an environment, only replacing free variables when they were needed. The most natural way to fix the error was to apply the substitution to the current expression immediately, removing the need to keep the substitution as an environment. This also removed an unnecessary distinction in the formalisation: in the original formulation the expression/environment pair $\langle e,\ \sigma \rangle$ behaved equivalently to the pair $\langle e[\sigma],\ \sigma \rangle$, yet the two were distinct. And finally, the use of Agda had a positive effect on the formulation of the representation of substitutions. In order to ensure totality in Agda we had to parameterise substitutions with the set of variables used in their domain and result. Far from being a hindrance, this led us to the monadic formulation of composition.

8 Extending the Language

In this paper we focused on a minimal language to emphasise the key elements of the reach problem and a solver based on lazy narrowing semantics. However, our results also scale up to a more realistic language that includes function application, lambda abstraction and fixed points [4]. This section briefly describes the changes that are required to the Agda formalisation.

First of all, the expression grammar is extended to include the three new constructors: function application, lambda abstraction and fixed points. To avoid ill-formed expressions the addition of these language features requires the new expression grammar to be typed. Therefore a function type is added to the language, along with the type of natural numbers. The small step semantics is extended to account for the new language constructs.

Our formalisation of the lazy narrowing semantics for the extended language restricts free variables, and by extension narrowing, to natural numbers. Although this is certainly a limitation, it is standard in the lazy narrowing literature, where a narrowing theory is generally described for first-order data initially, and then potentially extended to the higher-order case in subsequent work. With this restriction, the alteration to the lazy narrowing semantics and correctness proof is minor. The suspension predicate, $e \multimap x$, has to be updated as an expression can now be suspended within a function application or a fixpoint expression. We defined the lazy narrowing semantics by lifting the original semantics, and this definition remains unchanged except that we now lift the extended semantics. Finally, the lemmas, particularly the lift and unlift Lemmas (2 and 3), need updating to account for the additional cases. The proof of soundness and completeness remain identical under the updated lemmas.

The ease of this extension suggests it may be possible to generalise the theory by abstracting away from the details of the underlying language and semantics that is used, which is an interesting topic for further work.

9 Related Work

There is a large body of work on the theory of lazy narrowing in functional logic programming. We introduce and compare two particularly relevant theories to ours. In their seminal work, Antoy et al. [2] established the soundness and completeness of the related notion of needed narrowing, and the optimality of needed narrowing within a restricted domain. However, whereas our formalisation is based on extending a small-step semantics, theirs is based on classical rewrite systems. As a result, our approach is easier to mechanically verify, which we have done, as the semantics of our language has a direct representation in proof assistants. In fact, to the best of our knowledge, this is a first time that a lazy narrowing formalisation has had such a verification.

A formulation of lazy narrowing which is more closely related to ours is given by Albert et al. [1] in which a "natural" big-step semantics is defined before an implementation driven small-step semantics is introduced. Both semantics are call-by-need, implement sharing, and are proved to be equivalent. They go on to extend the small-step semantics with additional features such as equational constraints and external functions. There is a difference in motive in comparison to our work, as they establish lazy narrowing as a programming language feature whereas we are interested in using lazy narrowing to analyse the operation of a program. The difference manifests itself in the theories: they relate their small-step semantics back to their defining big-step semantics, whereas we relate our lazy narrowing semantics back to the underlying functional semantics.

10 Conclusions and Future Work

In this article we established the correctness of a reach solver for a minimal language, based upon a soundness and completeness result for a lazy narrowing semantics. Our final formulation of the semantics is the result of several iterations and improvements, and captures the main ideas of lazy narrowing in a simple and concise manner. In particular, the use of an underlying small-step semantics was instrumental in simplifying the theory. The simplicity along with the use of precise types enables a direct translation of our result to the Agda system [4]. There are number of interesting directions in which the theory developed in this article could be extended and improved, which are summarised below.

Other Reach Solvers. The work in this article lays the ground for attempting to formalise alternative and more general reach solvers, such as the Backward Reach solver defined in Naylor's thesis [13]. In addition, tools such as Lindblad's data generator [10] and Lazy SmallCheck [16] define logical *or* operators that evaluate both argument expressions in parallel, which could significantly improve the performance of lazy narrowing as expressions of the form e *or* e' can be reduced to true if either argument reduces to true in the current substitution state. We could easily add such an operator to our language. However, our formulation suggests a generalisation to this idea, in the form of evaluating branches in parallel and utilising equational reasoning on case expressions.

Other Language Features. We used a minimal language for simplicity, but it is important to consider how our approach generalises to other language features. For algebraic datatypes, we expect it should be straightforward to extend our theory using ideas from generic programming as in [8], while first-order functions could be handled by representing functions using tries as in the improved Lazy Smallcheck [15]. Another interesting area to explore is dependent type theory. Lazy narrowing is often used in automated property based testing and dependent type theory seems a natural coupling as it offers an inbuilt language for specifications. In this area there is also potential for interesting comparison to related work such as automated proof search [14].

Efficiency. We showed that the lazy narrowing definition of reachability for our language is correct with respect to the original specification of reachability. However we have not made any formal argument regarding the *efficiency* of the lazy narrowing approach, either against an alternative narrowing semantics or a naive approach based on brute force search. Such an argument could be made on the basis of simply counting the number of reduction steps required, or adopt a more sophisticated approach, for example using the idea of *improvement theory* [11], which has recently been used to prove that a general purpose optimisation technique for lazy languages never makes programs worse [6].

Acknowledgements. We would like to thank members of the FP Lab in Nottingham and the anonymous referees for useful comments and suggestions regarding this work.

References

1. Albert, E., Hanus, M., Frank, H., Oliver, J., Germán, V.: Operational semantics for declarative multi-paradigm languages. J. Symbolic Comput. **40**(1), 795–829 (2005)
2. Antoy, S., Echahed, R., Hanus, M.: A needed narrowing strategy. J. ACM **47**(4), 776–822 (2000)
3. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (2000)
4. Fowler, J.: Towards a Theory of Reach - Agda Proof (2015). https://github.com/JonFowler/theoryofreach
5. Gill, A., Runciman, C.: Haskell program coverage. In: Proceedings of the ACM SIGPLAN Workshop on Haskell (2007)
6. Hackett, J., Hutton, G.: Worker/Wrapper/Makes It/Faster. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (2014)
7. Hanus, M.: A unified computation model for functional and logic programming. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (1997)
8. Hinze, R.: A new approach to generic functional programming. In: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2000)
9. Hughes, J.: QuickCheck: an automatic testing tool for Haskell (QuickCheck manual). http://www.cse.chalmers.se/~rjmh/QuickCheck/manual.html

10. Lindblad, F.: Property directed generation of first-order test data. In: Proceedings of the Eighth Symposium on the Trends in Functional Programming (2007)
11. Moran, A., Sands, D.: Improvement in a lazy context: an operational theory for call-by-need. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (1999)
12. Naylor, M., Runciman, C.: Finding inputs that reach a target expression. In: Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (2007)
13. Naylor, M.F.: Hardware-Assisted and Target-Directed Evaluation of Functional Programs. Ph.D. thesis, University of York (2008)
14. Norell, U.: Towards a Practical Programming Language Based on Dependent Type Theory. Ph.D. thesis, Goteborg University (2007)
15. Reich, J.S., Naylor, M., Runciman, C.: Advances in lazy SmallCheck. In: Hinze, R. (ed.) IFL 2012. LNCS, vol. 8241, pp. 53–70. Springer, Heidelberg (2013)
16. Runciman, C., Naylor, M., Lindblad, F.: SmallCheck and lazy SmallCheck automatic exhaustive testing for small values. In: Proceedings of the First ACM SIGPLAN Symposium on Haskell (2008)

Functional Testing of Java Programs

Clara Benac Earle and Lars-Åke Fredlund(✉)

Babel Group, Universidad Politecnica de Madrid, Madrid, Spain
{cbenac,lfredlund}@fi.upm.es

Abstract. This paper describes an approach to testing Java code using a functional programming language. Models for Java programs are expressed as Quviq Erlang QuickCheck properties, from which random tests are generated and executed. To remove the need for writing boilerplate code to interface Java and Erlang, a new library, JavaErlang, has been developed. The library provides a number of interesting features, e.g., it supports automatic garbage collection of Java objects communicated to Erlang, and permits Java classes to be written entirely in Erlang. Moreover, as the library is built on top of the Erlang distributed node concept, the Java program under test runs in isolation from the Erlang testing code. The chief advantage of this testing approach is that a functional programming language, with expressive data types and side-effect free libraries, is very suited to formulating models for imperative programs. The resulting testing methodology has been applied extensively to evaluate student Java exercises.

Keywords: Software testing · Erlang · Java

1 Introduction

This paper describes a methodology for testing Java code which uses the property-based random testing tool Quviq Erlang QuickCheck [1], henceforth abbreviated as QuickCheck. As this version of QuickCheck provides no special facilities for testing Java code, and there are versions of QuickCheck adapted for Java (e.g., Java QuickCheck [2], JUnit with QuickCheck parameters [3], and ScalaCheck [4]), it might seem surprising that we choose to use a QuickCheck which has Erlang as its basis. The reasons are twofold: the Erlang based version of QuickCheck is a mature tool, especially compared with Java QuickCheck, and more importantly, we consider Erlang to be a much better *modelling* language than Java or Scala.

To use QuickCheck to check Java code is in practice not very difficult but rather tedious. The Erlang standard library Jinterface permits communication between Erlang and Java, but unfortunately a lot of Java boilerplate code has to be written for each Java program tested. To eliminate this source of testing inefficiency we designed a new Erlang library JavaErlang [5] on top of Jinterface, which removes the need to write boilerplate code. Moreover, as the library is built on top of the Erlang distributed node concept, the Java program under test runs in isolation from the Erlang testing code.

M. Serrano and J. Hage (Eds.): TFP 2015, LNCS 9547, pp. 40–59, 2016.
DOI: 10.1007/978-3-319-39110-6_3

The main contributions of this work are the new library JavaErlang, and a methodology for testing Java programs using QuickCheck. The JavaErlang library provides a number of interesting and novel features, e.g., it offers an Erlang programmer a seamless interface to Java code, permitting Java classes to be written entirely in Erlang, and enables safe automatic garbage collection of Java objects communicated to Erlang. The testing methodology is based on modelling the behaviour of Java programs and libraries using QuickCheck state machines, a natural approach since most of Java code is stateful. The methodology has been used extensively to test Java code developed by students at the Universidad Politecnica de Madrid, and has proven successful. The article summarises the experiences gained from applying the methodology to the task of testing a set of implementations of a medium-sized Java library. A number of interesting bugs were found, leading us to realise that innocent looking single-threaded Java programs can behave non-deterministically, due to the data structures used.

In the following a basic knowledge of the Erlang programming language is assumed. Section 2 summarises related work, and Sect. 3 discusses the new Java interface library. Section 4 explains the details for how models for Java programs are developed in Erlang, and the task of testing a larger API for finite state automata is used to demonstrate that the approach scales. Finally Sect. 5 summarises the results, and outlines issues for further work.

2 Related Work

There are a large amount of libraries and tools available that attempt to ease the task of interfacing functional programming languages with imperative or object-oriented programming languages such as e.g. Java. As an example, historically there have been several Haskell libraries available for interfacing with Java code, most of them bridging the two languages using the JNI (Java Native Interface) program framework (a foreign function interface). Examples of libraries using such an approach include the java-bridge package, GCJNI, and foreign-jni, with varying levels of maturity. Most of these efforts require the (automatic) generation of glue code in order to access a particular Java library.

Another approach to interfacing a functional programming language with Java is represented by Erjang [6], which is an implementation of an Erlang virtual machine running under Java. Although technically impressive, Erjang does not yet support full Erlang, and moreover, from the point-of-view of testing Java software using Erlang, it is not clear that such a close integration of the two languages is desirable.

A third approach to interfacing a functional programming language with Java is illustrated by the Erlang Jinterface library provided with Erlang. The library in essence provides the following functionality: (i) support for implementing a Java node which can connect with Erlang (distributed) nodes, permitting Erlang processes to communicate with the Java node, and (ii) a facility for the encoding of Erlang data terms as Java objects. Since this article is using Jinterface as an essential building block, it is worthwhile to examine the library in further detail.

To enable communication between an Erlang process and a Java node the Erlang process should send a normal Erlang term t, which Jinterface presents to the Java node as a Java object having the same term structure as the Erlang term t. That is, Jinterface provides Java classes corresponding to Erlang atoms (e.g., the Java class `OtpErlangAtom`), integers, lists, tuples (the class `OtpErlangTuple`), etc. A Java node that wishes to send a message to an Erlang process should encode the message using the above Java classes for Erlang data types. Thus the Erlang processes communicate using normal Erlang terms, while the Java node works with Erlang terms encoded as Java objects.

As an example, suppose an Erlang process wants the Java node to invoke a static method (a method that is not executed in the context of an object, but in the context of a class) `m(10)` in a class `C` we could let Erlang send a message {`'call_C_m', 10`}, and have the Java node execute the following code to interpret and execute the request:

```
OtpErlangObject msg = msgs.receive();
if (msg instanceof OtpErlangTuple) {
  OtpErlangTuple tup = (OtpErlangTuple) msg;
  if (tup.arity() == 2) {
    OtpErlangObject tag = tup.elementAt(0);
    OtpErlangObject arg = tup.elementAt(1);
    if (tag instanceof OtpErlangAtom) {
      String stag =
        ((OtpErlangAtom) tag).atomValue();
      if (stag.equals("call_C_m")) {
        int i = ((OtpErlangInt) arg).intValue();
        C.m(i); // Finally call the method
      } else ...
    } else ...
  } else ...
} else ...
```

Clearly there are a number of disadvantages to this approach. First, there has to be a prior agreement between Java and Erlang on a shared vocabulary (typically communicated as tuples as seen in the example above). This normally forces a Jinterface user to write a lot of boilerplate Java code. Secondly, while it is clear how to communicate primitive Java values (and arrays) to and from Erlang, how are Java object references to be communicated? Thirdly, compared to the earlier JNI based solutions, and Erjang, performance is lacking.

The QuickCheck random testing tool [7] has had a very positive impact on the use of functional programming for model-based testing. The Haskell based QuickCheck, and its many derivatives for other functional programming languages, are nowadays being used to model a large variety of software and hardware systems. As an example of successful cross-language model-based testing, Quviq is using their Erlang based QuickCheck tool [1] to test automotive software written in C [8].

3 The JavaErlang Library

The JavaErlang library[1] was born to address a number of limitations of Jinterface. JavaErlang is build on top of Jinterface, but provides more convenient mechanisms for enabling Erlang processes to interact with Java. The principal advantages of JavaErlang are:

- All public Java methods and fields of classes and objects are accessible without the need to prepare referenced Java classes in any way. A common requirement in many other language bridging libraries is to e.g. analyse the interfaces to classes offline, in order to automatically derive code to access class members.
- No Java code has to be written in order to use the library.
- Java object references can be freely communicated to and from Erlang processes. Moreover, Java objects whose references have been communicated to Erlang are still subject to safe automatic garbage collection.

As a first example, the following is a simple usage of the library (invoked from the Erlang shell):

```
> {ok,NodeId} = java:start_node().
{ok,100}
> I1 = java:new(NodeId,'java.lang.Integer',[10]).
{object,0,<<>>,0,100}
> I2 = java:new(NodeId,'java.lang.Integer',[10]).
{object,1,<<>>,0,100}
> java:call(I1,equals,[I2]).
true
> java:call
  (I1,equals,
   [java:new(NodeId,'java.lang.Integer',[7])]).
false
```

In the example we first start a new Java node (i.e., a Java runtime), and let Erlang connect to that node. Then, two Java Integer objects are created on the Java node, both initialised to the integer value 10. In Java this corresponds to calling the `Integer(int value)` constructor of the `Integer` class. The first call to `java:new` returns a tuple `{object,0,<<>>,0,100}` corresponding to the representation in Erlang of a Java object reference. Next, we check using the `Integer` class instance method `equals` that the first integer is equal (according to the Java `equals` method) to the second. Finally we create a third integer, initialised to 7, and check that it is not equal to the first.

Note that the example is entirely self-contained. There is for instance no need to analyse, or compile, the Java libraries before they become usable from Erlang. Neither does the example assume that a Java interpreter has already been started; the Java runtime is started using the call to `java:start_node()`. Third-party libraries are automatically made available too (as long as Erlang is informed of the location of these libraries).

[1] Available at: https://github.com/fredlund/JavaErlang.

Below we summarise the main functions of the `java` module API, assuming that `NodeId` is a Java node identifier, `Class` is the Erlang atom corresponding to a Java class, `Method` is an Erlang atom naming a method, and `Field` is an atom naming a field:

```
start_node() → {ok,NodeId}
new(NodeId,Class,[Val]) → Obj
call(Obj,Method,[Val]) → Val
call_static(NodeId,Class,Method,[Val]) → Val
get(Obj,Field) → Val
set(Obj,Field,Val) → Val
get_static(NodeId,Class,Val) → Val
set_static(NodeId,Class,Field,Val) → Val
```

`Object` is the representation of a Java reference in Erlang, and `Val` is an object reference, or an Erlang value that can be interpreted as a Java value (integers, Booleans, the atom `null`, ...). These functions are used to create a new instance of a class (`new`), to call an instance method of an object (`call`), to call a static (class) method (`call_static`), and functions for accessing and modifying object and class field (attribute): `get`, `set`, `get_static`, and `set_static`. A number of function variants are omitted from the above table, e.g., variants of the `call` functions identical to the above ones except that they accept an additional type argument which permits to elect the exact method invoked[2]. If such a type argument is missing, JavaErlang first calculates types for the method arguments, and uses those inferred types to select the appropriate method.

3.1 Internal Design

Using the Jinterface library, the JavaErlang library provides a Java node which listens for messages sent from Erlang nodes with information requests regarding classes (enumerating methods, fields and subclasses), and messages corresponding to requests to execute a method, or access or modify the value of a field. An Erlang node can connect to multiple Java nodes at the same time. The basic functionality of the library is to provide a set of mappings which associate Java object references with their representation as Erlang data values. That is, whenever a Java object reference should be returned to the Erlang node, it is mapped to its Erlang representation (a tuple), and vice versa, when the Erlang node communicates a Java object reference represented as an Erlang value, it is mapped to a real Java object reference in the Java node. Thus, there are two tables in the Java node:

$$\begin{aligned} \mathit{toErlang} &: \mathit{java.lang.Object} \mapsto \mathit{erlangRef} \\ \mathit{fromErlang} &: \mathit{erlangRef} \mapsto \mathit{java.lang.Object} \end{aligned}$$

[2] Multiple methods can share the same name in Java, but must have differently typed arguments.

An Erlang node keeps two tables, *java_classes*, a mapping from class names (an Erlang atom) to a class record with information about publicly accessible class members (methods, fields, etc.), and *java_class_ids*, a mapping from integers to class records.

The representation in Erlang of a Java object reference is a tuple:

$$\{\texttt{object}, \mathit{ObjectRefNo}, \mathit{RefCounter}, \mathit{ClassNo}, \mathit{NodeId}\}$$

henceforth referred to as an *Erlang object tuple*. The field *ObjectRefNo* is a unique integer identifying the object, *RefCounter* is a counter keeping track of the number of times this object reference has been communicated to the Erlang side (required for the purpose of garbage collection), *ClassNo* is an integer that identifies the Java class to which the object belongs, and *NodeId* identifies in which Java node the object resides.

The reflection programming technique [9] (provided in Java by the package `java.lang.reflect`) is used extensively in a Java node to obtain Java objects corresponding to Java classes, methods, etc. Note that these objects represent the actual classes and methods, rather than just their names. References to these objects, when converted into Erlang object tuples, can then be communicated to, and from, the Erlang node, to inform the Erlang node about the publicly accessible members of a class. To illustrate the design of the JavaErlang library, we below sketch the major steps in the implementation of the function `new(NodeId,ClassName,Args)`:

Class-lookup (Erlang)**:** upon invocation of the `new` function, the Erlang node first checks whether the class is already known to the Erlang node by looking it up in the table *java_classes*. If the class is not known, Erlang sends a message to the Java node identified by `NodeId` requesting information regarding the class, i.e., an enumeration of its public constructors, methods and fields. If the class is known, execution continues with the step "Inferring a constructor" below.

Class Reflection (Java)**:** upon receiving the request for class information Java proceeds to enumerate the constructors, etc., using reflection, obtaining Java object references corresponding to these class members, with associated type information for method arguments. Next, Java sends a reply to the Erlang side with the class information where object references are mapped into Erlang object tuples (using the map *toErlang* – see example step six below for a clarification), and with a unique class number (an integer).

Storing the class information (Erlang)**:** the Erlang node packages the class information into a record, and stores the class information for future use in the class table *java_classes* using the class name as key, and in the table *java_class_ids* as well, using the class number as key.

Inferring a constructor (Erlang)**:** next, Erlang proceeds to try to find a class constructor whose type matches the derived type of the actual parameters, using the information in the class record. As an option, a user can explicitly specify the desired constructor by supplying an extra parameter to the `new` function.

Sending a "call" message to Java (Erlang)**:** if a constructor is found, Erlang sends a message to Java node with the object tuple corresponding to the chosen constructor, the argument list, and the process identifier (pid) of the calling process, and waits for a reply message. If no constructor was found an exception is raised.

Executing the constructor (Java)**:** when Java receives the message containing the call request, it first finds a suitable thread in which to execute the request, or creates one if needed[3]. Java proceeds to map the constructor object tuple, and constructor arguments into Java object references (using the *fromErlang* map), and finally executes the constructor (using reflection).

Updating object reference tables (Java)**:** if the invocation of the constructor is successful, Java records an association between the new Java object reference and a fresh Erlang object tuple "`{object,ObjectId,0,ClassId,NodeId}`" (an *Erlang object tuple*) in the *fromErlang* and *toErlang* maps. `ObjectId` is chosen as a unique integer, and `ClassId` is a unique integer representing the object class. If the invocation fails, the Java side instead associates the resulting exception object with an Erlang object tuple.

Communicating the result to Erlang (Java)**:** the resulting object tuple (corresponding to a normal object reference, or the exception object) is sent back to the waiting process in the Erlang node using message passing.

Receiving the reply (Erlang)**:** the Erlang processes that called the constructor receives the reply, and examines the result. If the result is not an exception object it is simply returned as the result of the call to the `new` function, or, if the result was an exception, an exception is raised instead.

Figure 1 shows the flow of information exchanged between an Erlang node and a Java node when an instance of a class not previously known is created.

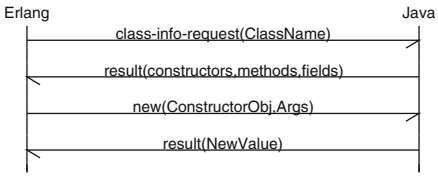

Fig. 1. Communications when creating a new Java object

The implementation of the `call` function, and `call_static` and the functions for accessing and modifying fields are analogous to the implementation of `new`, as described above. The `call` function must be invoked with an object tuple as the first argument. The Erlang node proceeds by first extracting the

[3] The library uses a 1-1 mapping between Erlang processes and Java threads, see Sect. 3.4.

class identifier from the object tuple, and looks it up in the *java_class_ids* table to obtain the class record. If no such class record exists, the Erlang node must query the Java node for the class information (in the same manner as is done in the step "Class-lookup" above). The rest of the implementation of `call` follows the pattern of `new`.

3.2 Boxing and Unboxing

To ease the task of working with Java in Erlang the JavaErlang library duplicates a number of tasks of a Java runtime, e.g., providing boxing and unboxing of primitive values and objects, providing functions for convenient access and creation of Java arrays, and also mapping Java primitive values to suitable Erlang primitive values.

As an example, the Java `null` value is represented as the Erlang atom `null`, and the Java Boolean values `true` and `false` are represented as the corresponding atoms `true` and `false`, and integer-like types and floating-point-like types are represented as normal Erlang integers and floats. When the heuristic type conversions fail, values can be explicitly type cast using the notation {`Type,Value`}. Java arrays can be constructed using the normal Erlang syntax for lists and strings. As an example, a Java short integer can be constructed using the syntax {`short,5`}, a Java character with {`char,$a`}, and a one-dimensional Java character array can be constructed with {{`array,char,1`},`"Hello World"`}.

3.3 Automatic Garbage Collection

As both Java and Erlang are languages with automatic garbage collectors, it is highly desirable that Java objects, whose references are communicated to Erlang, can still be garbage collected safely and automatically. However, the implementation of such a feature is not trivial.

As explained above, when a Java object reference is communicated to Erlang, the library establishes a mapping from the object reference to an Erlang object tuple, which should be preserved during the lifetime of the object (in two hash tables). The fact that object references are recorded in such tables, prevents premature garbage collection of Java objects whose references have been communicated to Erlang even if no data structure on the Java side any longer refers to them. However, to enable automatic garbage collection of objects communicated to Erlang the Erlang node must then be made to inform the Java node when an object tuple is no longer in use, so that the object reference can be removed from the tables. Unfortunately, Erlang does not have any language feature that would permit to implement this directly, e.g., there is no equivalent in Erlang to the Java `finalize` call (or Haskell "finalizers", e.g., as used by weak pointers). However, we have implemented (in C) such a feature for Erlang using the so called resource objects in the Erlang `erl_nif` foreign function interface library.

As a first step to enable automatic garbage collection, then, the map *toErlang* is extended to include the number of times an object reference has been communicated to Erlang: $\mathit{java.lang.Object} \mapsto \mathit{erlangRef} \times \mathit{nat}$. The reference counter is incremented whenever the same object reference is communicated to Erlang. Moreover, the Erlang object tuple,

$$\{\texttt{object}, \mathit{ObjectRefNo}, \mathit{RefCounter}, \mathit{ClassNo}, \mathit{NodeId}\}$$

which represents a Java reference, includes a natural number *RefCounter* which is incremented for each copy of the reference returned to Erlang. When the Erlang side receives the object tuple, it substitutes a fresh `erl_nif` resource object for the reference counter. Code on the Erlang side can now treat the reference object as a normal data value, and when no reference to the object reference containing the object resource remains, the `erl_nif` library will automatically call the "destructor" of the resource object. The destructor consequently sends a message to an Erlang process in the JavaErlang library which is responsible for garbage collection of Java objects, which forwards the message to the Java node. The java node can then decrement the reference counter in the *toErlang* map. If the counter becomes zero, the Erlang node no longer possesses any object tuples that refer to the Java object, and the mapping between object references and object tuples can then be safely removed from the *toErlang* and *fromErlang* tables, thus potentially permitting the Java object to be garbage collected (if there are no references to the object on the Java node).

There are a number of situations that require care in the above garbage collection scheme. First, we must ensure that object tuples cannot be garbage collected during calls from Erlang to Java. Consider for instance a call: `call(Obj1,equals,[Obj2])`. If, say, this is the last reference to the object tuple `Obj1` on the Erlang node then the object tuple could potentially be garbage collected after sending the call message to the Java node. However, the message regarding the garbage collection of `Obj1` could potentially be delivered at the Java node before the call message is delivered[4], thus removing the mapping from the table *fromErlang*. When the call message arrives, the object tuple can no longer be converted into a Java object reference, and the call simply cannot proceed. To protect against such race situations we keep a copy of all the object tuples that occur in a message sent from an Erlang node, until the Java node replies. Second, normal Erlang term equality can no longer be used to compare object tuples for referential equality, as two object tuples corresponding to the same Java object reference, but returned from different calls to Java, will not be syntactically equal. Instead, the function `java:eq(Obj1,Obj2)` should be used to compare object tuples for referential equality.

Note that this garbage collection scheme works even when object tuples are communicated between different Erlang processes, as the underlying `erl_nif` object resources used in their implementation are safe to communicate between processes.

[4] The message delivery guarantees for Erlang to Java communication are not well-defined. Moreover it is unwise to rely on such guarantees even in the case of pure Erlang-to-Erlang communication, see [10,11] for a detailed discussion.

3.4 Java Threads and Timeouts

By default the library maintains a 1-1 mapping between Erlang processes and Java threads, that is, each Erlang process will be serviced by the same unique Java thread. As a consequence the whole Java node is not blocked even if a particular method call is slow to execute. Such a process to thread mapping is moreover advantageous to interact with various Java libraries such as e.g. Swing (a GUI library), which require that API calls should always be executed by the same thread. This simply corresponds to, in our case, being called from the same Erlang process.

To enable to observe non-terminating Java method calls during testing, the library provides an optional timeout mechanism. If a timeout is specified, a Java method call will fail (raising an Erlang exception) if it does not return a result within the specified time limit.

3.5 Implementing Java Classes Using Erlang

With the capability to generate instances of arbitrary Java classes, and call arbitrary methods and access fields, clearly a large subset of Java APIs can be accessed using the library, without writing a single line of Java code. However, there are a few situations when a Java API requires a user to implement a new class, e.g., when the class implements a Java interface, and an instance of the class should be supplied as an argument to a method, or, when a concrete class must be supplied which extends an abstract Java class. Normal Java proxies (the `java.lang.reflect.Proxy` class) permits new classes to be created at runtime, but unfortunately does not permit to extend abstract classes. Instead, the JavaErlang library uses the Javassist byte code manipulation library [12] to permit the creation of new Java classes in Erlang.

The functionality provided by JavaErlang is explained here with a small example. In the Swing Java GUI, to react to a user pressing e.g. a button in a form, the programmer should register an object that implements the `Action` interface with the button to handle GUI events. This interface is implemented by e.g. the `AbstractAction` abstract class, which needs to be extended to be able to listen to events:

```
Button = java:new(N,'javax.swing.JButton',["Hello"]),
java_proxy:class
  (N, 'actListen', 'javax.swing.AbstractAction',
   [{{actionPerformed,['java.awt.event.ActionEvent']},
     fun actionPerformed/3}]),
Proxy = java_proxy:new(N,'actListen'),
java:call(Button,setAction,[Proxy]),
java:call(Pane,add,[Button]).
```

In the first line a new Swing button is created, and then a new "proxy" class `actListen` is created (i.e., a Java class implemented in Erlang) that redefines the method `actionPerformed` (with one argument of type

'java.awt.event.ActionEvent') whose implementation is provided by the Erlang function actionPerformed/3. In the third line and fourth lines an instance of the "proxy" class is created and passed as an argument to the method setAction of the button, and in the fifth line the button is added to a window pane. Finally the function actionPerformed is shown below.

```
actionPerformed(_, _State, Event) ->
  Button = java:call(Event,getSource,[]),
  Text = java:call(Button,getText,[]),
  String = case java:string_to_list(Text) of
             "Hello" -> "World";
              _ -> "Hello"
           end,
  java:call(Button,setText,[String]),
  {reply,void}.
```

The function is called with three arguments, the first context representing the object and method called, and the second argument corresponds to the state of the object in which the new actionPerformed resides. Java classes defined using Erlang will typically be stateful too, and hence the library provides a state that is remembered between invocation of class methods. The initial state may be supplied as an argument to java_proxy:new. The third, and following arguments, represent the actual arguments of the called method, in this case the event that took place. The function above modifies the label of the button (alternating between displaying "Hello" or "World depending on the text displayed on the button), and finally returns {reply,void} indicating that method does not return any value. The function java:string_to_list/1 converts a Java string to an Erlang list of characters.

The methods of such "Erlang objects" are always executed as if they were declared "synchronized", i.e., in mutual exclusion with other methods. Moreover, as a current limitation of the library, Java methods defined in Erlang cannot be recursive, i.e., if actionPerformed above were to call itself recursively, the recursive call would block indefinitely.

Internally the proxy facility is implemented on the Erlang side by having a dedicated proxy server process, to which the Java node sends a message when the redefined method of a proxy object is called. The Erlang proxy server keeps a proxy table which records the state of all proxy objects, i.e., whether an object is executing, its queue of outstanding calls, and the state of the proxy object. If a call message is received by proxy server, it looks up the proxy object record, and if the object is not currently busy, spawns a new Erlang process which executes the function call, with the current state of the proxy object as a parameter, and marks the proxy object busy in the proxy table. The proxy server then continues, possibly servicing calls to other proxy objects. If the object was busy, the call is added to the proxy object queue. When the spawned Erlang process has finished the execution of the function call, resulting in a return value and a new proxy object state, it sends a message to the proxy server. The proxy server eventually

receives this message, and as a consequence, it sends a message to the Java side, and then updates the state of the proxy object, and marks it as non-busy.

3.6 Limitations

The principal limitation of the library is the performance penalty incurred when issuing calls of Java methods from Erlang, compared to Java-to-Java calls. The overhead has multiple sources, e.g., the use of reflection, communication costs including marshalling of object references, the algorithm for determining which method instance to invoke, etc. Moreover the library uses additional memory, keeping two entries per object reference communicated to Erlang in the Java node, and in the Erlang node allocating one `erl_nif` object resource for every copy of an object reference communicated to Erlang. As a small benchmark, on a PC running Ubuntu 13.04 with an Intel(R) i7-2640M cpu at 2.80 GHz, around 3000 calls to Java can be made per second using the current version (1.3) of the library; several orders of magnitude worse than direct Java-to-Java calls.

A limitation of the library is that it is currently not possible to implement object locking in Erlang. Object locks implemented in Java (using e.g., the `synchronized` keyword) are respected, but there is currently no mechanism in JavaErlang to specify a custom locking protocol as this would require issuing and executing `monitorenter` and `monitorexit` byte code instructions. There are a number of possible solutions, e.g., to implement in JNI (Java Native Interface) methods corresponding to these byte code instructions, which would then be callable from Erlang, or directly manipulate byte codes using Javassist.

4 Testing Java Code Using QuickCheck

The basic functionality of QuickCheck is simple: when supplied with an Erlang data term that encodes a Boolean property, which may contain universally quantified variables, QuickCheck generates a random instantiation of the variables, and checks that the resulting Boolean property is true. This procedure is by default repeated at most 100 times. If for some instantiation the property returns false, or a runtime exception occurs, an error has been found and testing terminates.

4.1 QuickCheck State Machines

For checking "stateful" code, QuickCheck provides the `eqc_statem` library. Here the tested "object" is not a simple Boolean property, but rather a sequence of function calls each with an associated post condition that determines whether the execution of a call was successful or not. The library first *generates* a suitable test, i.e., a sequence of API calls, and then proceeds to *execute* the test, checking for each API call that the result was the expected one given the history of calls. Thus, the state machine acts as a *model* for the program under test.

To use the `eqc_statem` library a user has to supply a "callback" module with a set of functions with predefined names, which are called by QuickCheck during test generation *and* test execution.

```
initial_state()
command(State)
precondition(State,Call)
next_state(State,Result,Call)
postcondition(State,Call,Result)
```

A test state is kept both during test generation and test execution. The state is initialised by the `initial_state` function. API calls are generated by the function `command`, which returns symbolic calls of the form `{call,ModuleName,Function,Args}`. The `next_state` function is called both during test generation and test execution to modify the model state; the `Result` parameter is a symbolic variable during test generation, whereas it contains the actual call result during test execution. Hence, a state is typically symbolic during test generation, and concrete during test execution. The `postcondition` function checks that the return value of a call is correct.

4.2 Testing a Large Java Library

In this section we detail the techniques used to verify a larger Java library, and summarise the lessons learnt.

As an obligatory part of a course on algorithms and data structures, students at the Universidad Politecnica de Madrid were tasked with implementing a finite automata library in Java. In total around a 120 students were attending the course, and each student had to hand in his/her own solution. The requirements were to implement basic operations for both deterministic and non-deterministic (with epsilon transitions) automata, e.g., methods for constructing and decomposing automata, for computing the next automaton state provided the current state and a label as arguments, and to check whether a string (a sequence of labels) is accepted by an automaton. Moreover, students were tasked with implementing the following more complex functionalities: minimising a deterministic automaton, converting a regular expression into a non-deterministic automaton (with epsilon transitions), and converting a non-deterministic automaton into a deterministic one.

To save correction time, clearly a strong motivation for applying (property-based) testing existed in this case, and so a model for the automata library was developed using QuickCheck and JavaErlang.

Model. The model developed is a monolithic one, meaning that only a single `eqc_statem` state machine is specified, handling in total 20 different API calls, ranging from basic operations such as adding a transition to a finite state automaton, to minimising an automaton. To check the correctness of the API calls, the model must naturally implement the same functionality as the API (or use libraries to do so); e.g. the QuickCheck model for finite state automata also contains code for minimising a deterministic automaton. In the following a number of important facets of the test model are highlighted.

Model State. The state of the model is an Erlang record storing the finite automata created, the states created (the API permits free states not assigned to an automaton), and a set of testing options (see the Section *Parametricity* below). An automaton is composed of a number of states, a number of transitions, has an initial state, may be deterministic, can be explicit (was constructed using the low-level API), and may have been minimised. A state can be fresh (not part of an automaton), and may be accepting.

Parametricity. To be able to focus attention during testing on different parts of the whole API, the generation of a particular API call is made conditional on a set of testing parameters, which are initialized at the start of testing. These parameters are added to the state machine state, and checked when an API call is generated. As an example, a call to the method `addTransition` in the class `DFA` (a deterministic automaton) is generated by the following clause in the function `command(State)`:

```
[{call,?MODULE,addTransition,
  [FAId,FAState1,label(),FAState2]} ||
    test_dfa(State),
    test_basic_constr(State),
    FAId <- explicit_dfa_ids(State),
    FAState1 <- states_in_fa(State,FAId),
    FAState2 <- states_in_fa(State,FAId)]
```

The clause will generate a list of possible calls to add a transition to an automaton identified by `FAId`, from state `FAState1`, with a randomly generated label `label()`, to state `FAState2`, *if* the deterministic part of the API is being tested (`test_dfa(State)`), and *if* basic API calls are tested (`test_basic_constr(State)`). Moreover, `FAId` must be an existing deterministic automaton, and `FAState1` and `FAState2` must be states in that automaton. For completeness the Erlang functions that actually invokes Java to create a new transition are shown below:

```
addTransition(FAId,FAState1,Label,FAState2) ->
  java:call
    (FAId,addTransition,
     [ensure_object
      (java:new(node_id(),'automata.Transition',
        [FAState1,create_label(node_id(),Label),
        FAState2]))]).

create_label(NodeId,Label) ->
  ensure_object(java:new(NodeId,'automata.MyLabel',
     [java:list_to_string(NodeId,Label)])).

ensure_object(Object) ->
  case java:is_object_ref(Object) of
```

```
    true -> Object;
    false -> throw(ensure_object)
  end.
```

The function `ensure_object` verifies that its argument is a valid object reference, and not the null value, nor an exception. If the argument is not valid an exception is thrown; otherwise the argument is returned. Although not strictly needed, the function greatly improves error diagnostic by identifying the exact point at which e.g. a null pointer was returned when a chain of Java calls are executed (as is the case for `addTransition`).

Creating Complex Finite Automata. Normally, finite state automata are constructed using the rather cumbersome API available: creating an automaton, creating states, creating transitions using these states, and finally adding such transitions to the automaton. The frequency of generating these operations are controlled by weights in the QuickCheck model. Still, even when weights are carefully chosen, it is clear that the possibility of generating complex automata for testing is rather low when using only such "low-level" operations. Thus, as a configurable alternative, to generate more complex finite automata with a higher probability, automata can also be generated by first generating a random regular expression, and then translating the regular expression into a non-deterministic automaton, which is optionally made deterministic.

Handling Exceptions. Exceptions in the Java code are treated as normal return values (tagged with the atom `java_exception`). Below a small excerpt is shown from a function that checks whether the return value of a call matches the expected return value (called from `postcondition`):

```
expect_eq(Value,Result) ->
  case {Value,Result} of
    {{exception,ClassName}, {java_exception,Obj}} ->
      java:instanceof(Obj,ClassName);
    _ -> ...
  end.
```

If the call returned an exception, and an exception was expected, the code checks that the returned exception is a correct exception instance.

Handling Freshness and Absence of Change. When implementing a Java method `compose`, in a class `SomeClass`, of the shape:

```
public class SomeClass {
  public SomeClass compose(SomeClass s) {
    // compose this and s somehow,
    // return the result as a fresh instance
  }
}
```

which is supposed to return a fresh object, our students commonly make the mistake of reusing either the object instance itself, or the argument instance, instead of creating a new object. For such methods the requirement on returning fresh object instances is captured by including a call to the function `is_fresh` in the `postcondition` function:

```
is_fresh(Object,State) ->
  lists:all(fun (KnownObject) ->
                not(java:eq(Object,KnownObject))
            end,
            all_objects(State)).
```

The function `all_objects` returns a list of all the previously created objects known to the model, `java:eq` checks referential equality for Java object references sent to Erlang. Moreover, checking that an instance method does not have any side effects, i.e., that the method does not change the fields in its object or class, nor any other object or class, is automatic thanks to the manner in which random testing is implemented by `eqc_statem`. Side effect free methods are declared not to change the model state simply by including a default clause in `next_state`:

```
next_state(State,Var,Call) ->
  case Call of
    ...;
    _ -> State   %% for methods without side effects
  end.
```

QuickCheck will automatically generate tests that contain calls to such methods, followed by calls to other methods, and will check the postconditions of these method calls, thus detecting whether the supposedly side-effect free method was truly side-effect free.

Bugs Found Using Property-Based Testing. While it is clear that the majority of bugs found in the automata libraries were related to mistakes in handling boundary conditions (e.g., failing to correctly handle empty finite automata, etc.) a number of more interesting bugs were found during testing with JavaErlang and QuickCheck. We include a small survey of these bugs here, for a number of different reasons. First, their detection by QuickCheck is a convincing argument of the value of random testing in our opinion – we are not sure that they would have been detected by a more standard approach of focusing on boundary condition testing, or using coverage based testing techinques such as e.g. MC/DC [13]. Moreover, a number of these bugs illustrate nicely the difficulties of using an imperative language. Another set of bugs illustrate another type of difficulty, shared between imperative and functional programming languages: working with complex data structures can lead to program errors that are difficult to debug. In fact, using such data structures can even transform

a completely deterministic program into a non-deterministic one, whose behaviour depends crucially on the execution environment. Finally, in some cases the detection of a bug was largely due to luck. Studying the bug provides insights in how to improve testing in order to make bug detection more probable.

Using "==" instead of "equals". Many students used the "==" operator, which checks referential (pointer) equality, even though the objects compared were not created by them. A correct solution would have been to use the `equals` method instead. Using "==" led to failures for instance when comparing transition labels that were semantically equal but were created at different times.

Using Iterators. Most students used a Java `HashSet` to store e.g. the states of an automaton. Unfortunately using this data structure correctly turned out to be rather difficult. To compute the reachable states of a non-deterministic automaton typical student code did the following:

```
HashSet<State> states = ...;
for (State st : states)
  states.addAll(epsilonClosure(st));
```

That is, the code iterated over the states already in the set, and for each state, added also the states reached by a closure operation. From a functional programming point-of-view the code looks pretty clear and correct.

However, nondeterministically, such code failed during testing with an exception `Concurrent-Modification-Exception`. This was rather surprising to the students, as all student programs were single threaded. The Java documentation[5] for that exception states that: it is not permissible for one thread to modify the data structure while another thread is iterating over it. Moreover, if a single thread modifies a collection directly while it is iterating over it, this exception may be thrown.

Clearly there was no attempt to access the same collection from different threads. However, as the second sentence states, this exception can also be raised by a single thread. In our opinion it is a rather unfortunate state-of-affairs that Java raises this exception, as it prevents a natural style of programming. Solutions advocated in Java literature range from adding elements to temporary data structures, or making a copy of the data structure before iterating, to using the add/remove operations supplied by iterators.

To improve the detection rate for this type of errors each test case was repeated a number of times using the QuickCheck macro `?ALWAYS(N,Prop)`, which is equivalent to the property `Prop` except that it is tested `N` times (or until it fails). This is similar to how concurrent programs are tested under QuickCheck.

Modifying Elements Stored in a Hashtable. It was quite common student practice to first store a state in a hashtable, *and then modify it*, and finally do a look-up in the hashtable using the object. An example error:

[5] http://docs.oracle.com/javase/6/docs/api/java/util/ConcurrentModificationException.html.

```
HashSet<State> states = ...;
State st = new State();
st.accepting = false;
states.add(st);
// ...
st.accepting = true;
// ...
if (states.contains(st)) ... // wrong
```

This code typically fails to locate the state in the hash table, as its hash code is computed with the help of the field `accepting` which was `false` at the time of storage, but `true` when the hash table is searched.

Non-deterministic Data Structures are Difficult. As a surprise, even though the student solutions were thought to be totally deterministic, and generated test cases certainly are, running the same test case on the same student solution sometimes failed, and sometimes succeeded.

The reason turned out to be related to the implementation of the hash table data structure in Java. An example error:

```
boolean found = false;
State selected;

for (State st : states)
  if (!found && someConditionOnState(st)) {
    found = true; selected = st;
  }

// selected was used here to compute some result...
```

It turned out that the state table contained a number of states meeting the condition `someConditionOnState`, but the condition was not sufficiently strict, leading to potentially selecting a bad state, which causes a later test failure.

For the hash table data structure the order of elements returned by an iterator depends on the hash code of an element. In this case the hash code of a `state` object was computed using the default method, which uses an integer related to the location of the object in the heap. As the heap location is not stable from one machine to another, in practice it turned out that the order in which elements were returned by the iterator varied from one machine to another.

So, when testing on some machines, an incorrect element was selected first, leading to a test failure, while on other machines, a correct element was selected first, leading to test success.

Fixing this type of (testing) problem is nontrivial, as we cannot really hope to change the organisation of the heap by e.g. repeated testing using the QuickCheck macro `?ALWAYS(N,Prop)`. A solution would be to introduce an operation to randomise the heap before running each test case, and then repeating each test case a number of times.

Efforts. In total the automata model comprises around 1300 lines of Erlang code. While no scientific effort has been made to quantify the time and resources required to create these models the effort in our opinion has not been excessive. Creating state machine models is a learned skill, and since Java is a rather regular language, much model reuse is possible.

5 Conclusions

The main contributions of this work are the new library, JavaErlang, and a methodology for testing Java programs using Erlang QuickCheck.

The JavaErlang library removes the need to write Java code, and permits to focus on the task of designing models in Erlang, which in our opinion is a far superior modelling language compared to Java. Moreover, the library enables safe inter-language automatic garbage collection as a novel feature. Such a library can clearly be developed for other combinations of languages too, although to reduce the amount of programming required an analogue of the Erlang Jinterface library is needed. Moreover, a reflective capability is helpful.

The testing methodology is based on modelling the behaviour of Java programs and libraries using QuickCheck state machines, a natural approach since most of Java code is stateful. The methodology has proven successful, and has been used extensively to evaluate Java code at the Universidad Politecnica de Madrid.

We are currently exploring a number of items for future work. First, although the present work focuses on verification of non-threaded Java code, there are no intrinsic limits in the methodology that would prevent the testing of multi-threaded Java programs. In fact, the QuickCheck `eqc_statem` state machine library already does provide support for verification of atomic APIs, i.e., APIs whose operations may be interleaved in practise, but the *effects* of which can be explained using a serialisation of the operations [14].

A potential problem with using the JavaErlang library for testing concurrent Java code, is that since calling Java methods from Erlang is rather slow compared to Java-to-Java calls, some race conditions in the Java code may never be explored during testing with JavaErlang. However, early experiments with using the library for testing concurrent Java programs show promise [15]. Moreover, testing with QuickCheck can be combined with support techniques such as e.g. randomising the Java scheduler [16].

We would also like to explore the use of the JavaErlang library outside the domain of testing. One particularly interesting possibility would be to use Java GUI libraries for programming Erlang graphical applications. The benefits of such an approach compared to using the normal Erlang GUI library, which interfaces with wxWidgets, is primarily increased reliability (a buggy Erlang wxWidget GUI application can crash the Erlang node in which it is running), and having access to a more complete GUI, which would also be easier to maintain. The support for wxWidgets in Erlang is currently rather incomplete, and moreover, preparing a new release of wxWidgets for inclusion in Erlang requires a substantial effort.

Acknowledgement. Much of the work presented in this article has been realised in the context of the EU FP7 projects ProTest and Prowess, and we are grateful for many helpful suggestions received from the participants of these projects. Thanks are also due to the students at the Facultad de Informatica at the Universidad Politecnica de Madrid who helped refine the testing methodology by writing programs containing some truly interesting bugs.

References

1. Arts, T., Hughes, J., Johansson, J., Wiger, U.: Testing telecoms software with Quviq QuickCheck. In: Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, ERLANG 2006, pp. 2–10. ACM, New York (2006)
2. Jung, T.: Quickcheck for Java (2014). https://bitbucket.org/blob79/quickcheck. Accessed 30 June 2014
3. Holser, P.: Junit-quickcheck: quickcheck-style parameter suppliers for JUnit theories (2014). https://github.com/pholser/junit-quickcheck. Accessed 30 June 2014
4. Nilsson, R.: Scalacheck: property-based testing for Scala (2014). http://www.scalacheck.org/. Accessed 30 June 2015
5. Earle, C.B., Fredlund, L.: JavaErlang (2014). https://github.com/fredlund/JavaErlang. Accessed 30 June 2014
6. Thorup, K.K.: Erjang (2014). https://github.com/trifork/erjang. Accessed 30 June 2015
7. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of Haskell programs. SIGPLAN Not. **35**(9), 268–279 (2000)
8. Svenningsson, R., Johansson, R., Arts, T., Norell, U.: Formal methods based acceptance testing for autosar exchangeability. SAE Int. J. Passeng. Cars- Electron. Electr. Syst. **5**(1), 209–213 (2012)
9. Smith, B.C.: Reflection and semantics in a procedural language. Ph.D. thesis (1982)
10. Svensson, H., Fredlund, L.: A more accurate semantics for distributed Erlang. In: Proceedings of the 2007 ACM SIGPLAN Erlang Workshop, October 2007
11. Svensson, H., Fredlund, L.: Programming distributed Erlang applications: pitfalls and recipes. In: Proceedings of the 2007 ACM SIGPLAN Erlang Workshop, October 2007
12. Chiba, S., Nishizawa, M.: An easy-to-use toolkit for efficient java bytecode translators. In: Pfenning, F., Macko, M. (eds.) GPCE 2003. LNCS, vol. 2830, pp. 364–376. Springer, Heidelberg (2003)
13. Chilenski, J.J., Miller, S.P.: Applicability of modified condition/decision coverage to software testing. Softw. Eng. J. **9**(5), 193 (1994)
14. Claessen, K., Palka, M., Smallbone, N., Hughes, J., Svensson, H., Arts, T., Wiger, U.: Finding race conditions in Erlang with QuickCheck and PULSE. In: Proceeding of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP 2009, Edinburgh, Scotland, UK, pp. 149–160 (2009)
15. Fredlund, L.Å., Herranz, Á., Mariño, J.: A testing-based approach to ensure the safety of shared resource concurrent systems. In: Canal, C., Idani, A. (eds.) SEFM 2014 Workshops. LNCS, vol. 8938, pp. 116–130. Springer, Heidelberg (2015)
16. Park, C.S., Sen, K.: Randomized active atomicity violation detection in concurrent programs. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA, pp. 135–145 (2008)

Type Class Instances for Type-Level Lambdas in Haskell

Thijs Alkemade(✉) and Johan Jeuring

Utrecht University, Utrecht, The Netherlands
me@thijsalkema.de, J.T.Jeuring@uu.nl

Abstract. Haskell 2010 lacks flexibility in creating instances of type classes for type constructors with multiple type arguments. We would like to make the order of type arguments to a type constructor irrelevant to how type class instances can be specified. None of the currently available techniques in Haskell allows to do this in a satisfactory way.

To flexibly create type-class instances we have added the concept of type-level lambdas as anonymous type synonyms to Haskell. As higher-order unification of lambda terms in general is undecidable, we take a conservative approach to equality between type-level lambdas. We propose a number of small changes to the constraint solver that will allow type-level lambdas to be used in type class instances. We show that this satisfies our goal, while having only minor impact on existing Haskell code.

Keywords: Haskell · Type class · Type-level lambda · Higher-order unification

1 Introduction

The first version of the `unittyped` package [1] used a datatype similar to:

$$\textbf{data}\ \mathit{Value}\ v\ u\ d = \mathit{Value}\ v$$

A *Value* v u d contains an object of type v and is tagged with phantom types u and d. The type u represents the physical unit of the value (meters, miles, seconds, etc.) and d the dimension of that unit (length, time, etc.). Using type classes, u and d determine what operations may be done on these values, for example, only allowing addition when the dimension of the values is the same. After the first release, a feature request asked for a *Functor* instance for *Values*. *Functor* is a type class given by Fig. 1.

The only possible instance that the datatype would allow would give *fmap* the type:

$$\mathit{fmap} :: (a \to b) \to \mathit{Value}\ v\ u\ a \to \mathit{Value}\ v\ u\ b$$

M. Serrano and J. Hage (Eds.): TFP 2015, LNCS 9547, pp. 60–84, 2016.
DOI: 10.1007/978-3-319-39110-6_4

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

Fig. 1. The definition of *Functor*

This is not a useful instance: it can only change the dimension. Changing the dimension but keeping the same unit breaks the invariants the library is supposed to guarantee. The desired *fmap* instance would replace the v type argument:

$$fmap :: (a \to b) \to Value\ a\ u\ d \to Value\ b\ u\ d$$

However, Haskell doesn't make it possible to give this instance. Eventually, all uses of the *Value* type were rewritten to use the definition:

```
data Value u d v = Value v
```

A type class in Haskell is a set of polymorphic functions that can only be used on types that have instances for that class [5]. This way programmers can use the same name for similar functions. This allows for more concise notation and code that uses the type class can be re-used, requiring only new instances to be written. For example, the *Eq* class makes it possible to use $\equiv$ for any type that has an instance for *Eq*, instead of requiring many different functions for checking equality.

Type classes can not only be specified for normal types of kind $*$, but also for types of arrow kinds like $* \to *$. Every type class requires its instances to have a specific kind, determined by how many arguments the type variable receives in the function signatures of the type class [10]. For example, *Eq* has a type variable of kind $*$, as the type variable a in the class head does not receive any arguments:

```
class Eq a where
  (≡) :: a → a → Bool
```

The *Functor* class has a type variable of kind $* \to *$, as f is used with one argument (as a and b are of kind $*$). See Fig. 1.

One example of a type class using a type of kind $* \to * \to *$ is *Category* (Fig. 2).

```
class Category cat where
  id :: cat a a
  (∘) :: cat b c → cat a b → cat a c
```

Fig. 2. The definition of *Category*

The order of arguments for a function is usually selected based on what is the most convenient. For example, similar functions can be given similar orderings of arguments: the functions in `Data.Map` that receive a single *Map* argument receive the map as their last parameter. When a partially applied function is needed and the missing parameters are not the last arguments, then functions like *flip* and λ-abstractions can be used to rearrange the arguments to obtain any desired ordering of parameters.

Choosing the order of the type arguments of a type constructor may appear to be similar. The exact order is important when considering which higher-kinded types can be formed, but an equivalent of *flip* does not exist. For example, a type constructor with three arguments allows only three types of higher kinds to be formed:

$$\begin{array}{ll} \mathit{Value} & :: * \to * \to * \to * \\ \mathit{Value}\ v & :: * \to * \to * \\ \mathit{Value}\ v\ u & :: * \to * \\ \mathit{Value}\ v\ u\ d & :: * \end{array}$$

The higher-kinded type $\mathit{Value} \cdot u\ d$, of kind $* \to *$, where a type argument is substituted at the position of the $\cdot$, cannot be constructed in this way. This also means that, if a one class requires them to be in a certain order, but another class requires a different order, then it is impossible to give both instances at the same time.

In this paper we show how we can extend Haskell with a restricted form of type-level lambdas in class instance heads, so that we for example can write the following:

$$\begin{array}{l} \textbf{data}\ \mathit{Value}\ v\ u\ d = \mathit{Value}\ v \\ \textbf{instance}\ \mathit{Functor}\ (\Lambda v\ .\ \mathit{Value}\ v\ u\ d)\ \textbf{where} \\ \quad \mathit{fmap}\ f\ (\mathit{Value}\ v) = \mathit{Value}\ (f\ v) \end{array}$$

```
*Main> fmap (+1) (Value 42)
Value 43
```

In other words, the goal of this paper is to make the order of the arguments of a type constructor flexible. In particular, they should have no effect on how instances of higher-order classes can be defined. We require our solution to satisfy the following conditions:

1. The type checker requires no user-added type signatures where they are currently not required.
2. It should not be required to duplicate functions or type classes or to make large changes to existing functions or type classes.

The rest of this paper is organized as follows: Sect. 2 explains potential solutions to this problem using existing techniques and approaches, and the problems with these solutions. Section 3 formalizes the notion of type-level lambdas. Section 4 gives a general background about type checking and constraint solving in Haskell, and Sect. 5 gives the changes necessary to support type-level lambdas in GHC. Section 6 shows what is possible with these changes and Sect. 7 lists some potential problems with other GHC extensions.

2 Using Existing Concepts

This section discusses how existing concepts in Haskell might be used to solve the problem described in the previous section. None of our attempts solves the problem satisfactorily.

Currently, the simplest solution to obtain the correct instances for a type is to first consider the type classes that a type should have instances for, and then order the type variables accordingly. If the type variables of existing code do not match the order required for the desired class instances, they can be reordered to match by rewriting the type everywhere it is used. For example, if *Value v u d* later needs to have *Functor* instance that works on v, that would mean replacing every *Value v u d* with *Value u d v*. In a large codebase, this could be a significant amount of work.

2.1 Type Synonym Instances

The GHC extension `TypeSynonymInstances` [11] may appear to be a good solution. This extension allows type synonyms in the head of an instance declaration. Without this extension, only newtypes and datatypes can be used in the class instance heads.

By creating a type synonym that orders the type variables in the way they should be used by that class, the desired type for the instance could be specified. For example, we can create a new type synonym for *Value* which specifies the order of the type variables for its *Functor* instance.

```
data Value v u d = Value v
type ValueFunctor u d v = Value v u d
instance Functor (ValueFunctor u d) where
  fmap :: (a → b) → Value a u d → Value b u d
```

As nice as this may seem, it will not be accepted by GHC: `TypeSynonymInstances` only allows fully applied type synonyms in the instance head. A type synonym cannot be partially applied to form a type of kind $* \to *$ and be supplied to *Functor*. We will look further into this restriction in Sect. 3.1.

The only way a type synonym can be used to represent a type of kind $* \to *$ (or higher), is when the rhs of the type synonym already has kind $* \to *$:

```
data Value v u d = Value v
type ValueFunctor y x = Value v y
```

But this implies we are back at supporting only the limited set of higher-kinded types given in Sect. 1. This solution does not meet our main goal.

2.2 More Type Classes

Another solution would be to add more type classes. For every ordering of type variables a programmer might want to use for a class, a new copy of the class is added. The `Bifunctors` package [7] uses this approach: it allows types to be specified as functors on both the last and the second to last variable at the same time. For example, we can create *Functor*-like classes using the second and third variable with:

```
class Functor2 f where
  fmap2 :: (a → b) → f a x → f b x
class Functor3 f where
  fmap3 :: (a → b) → f a x y → f b x y
```

The advantage of this solution is that instances for *Functor2* and *Functor3* do not overlap. For every variable of a type constructor it is possible to indicate whether or not it allows an $fmap[n]$.

There is however a serious disadvantage to this solution: every function with a *Functor* constraint needs to be copied for *Functor2*, *Functor3*, etc. The implementation will be the same, except for the use of *fmap2*, *fmap3*, etc. instead of *fmap*:

```
increase :: (Functor f) ⇒ f Int → f Int
increase = fmap (+1)
increase2 :: (Functor2 f) ⇒ f Int x → f Int x
increase2 = fmap2 (+1)
increase3 :: (Functor3 f) ⇒ f Int x y → f Int x y
increase3 = fmap3 (+1)
```

Where the main goal of type classes is to avoid code duplication, this solution adds code duplication. Another disadvantage of this solution is that the number of extra type classes increases fast, especially for even higher- order type classes. For example, the *Category* class requires type constructors of kind $* \to * \to *$ (Fig. 2). Every possible pair would require a separate class, *Category_2_3*, *Category_1_3*, *Category_2_1*, etc.

While this solution meets our main goal, it does not satisfy the second condition: existing functions using type classes cannot be reused for the newly introduced type classes.

2.3 Newtype Wrappers

The `TypeCompose` package [3] contains the newtype definition:

$$\textbf{newtype}\ \mathit{Flip}\ t\ b\ a = \mathit{Flip}\ \{\mathit{unFlip} :: t\ a\ b\}$$

Flip can be viewed as a type-level variant of *flip*: the last two type arguments of the *Flip* type constructor are the last two type arguments of the wrapped type, but swapped. This also makes it possible to write instances where the last two variables are swapped:

$$\begin{aligned}
&\textbf{data}\ \mathit{Value}\ u\ v\ d = \mathit{Value}\ v\\
&\textbf{instance}\ \mathit{Functor}\ (\mathit{Flip}\ (\mathit{Value}\ u)\ d)\ \textbf{where}\\
&\quad \mathit{fmap} :: (v \to v')\\
&\qquad \to\ \mathit{Flip}\ (\mathit{Value}\ u)\ d\ v\\
&\qquad \to\ \mathit{Flip}\ (\mathit{Value}\ u)\ d\ v'\\
&\quad \mathit{fmap}\ f\ (\mathit{Flip}\ \{\mathit{unFlip} = \mathit{Value}\ v\})\\
&\qquad\qquad = \mathit{Flip}\ \{\mathit{unFlip} = \mathit{Value}\ (f\ v)\}
\end{aligned}$$

It is not only possible to flip the last two arguments, but *Flip* can be generalized to every reordering of type variables:

$$\begin{aligned}
&\textbf{newtype}\ \mathit{Flip2}\ t\ c\ b\ a\ \ \ = \mathit{Flip2}\ \{\mathit{unFlip2} :: t\ a\ b\ c\}\\
&\textbf{newtype}\ \mathit{Flip3}\ t\ d\ b\ c\ a = \mathit{Flip3}\ \{\mathit{unFlip3} :: t\ a\ b\ c\ d\}
\end{aligned}$$

The difference with `TypeSynonymInstances` is that *Flip*s are newtypes, not type synonyms. Therefore the type on the instance head is different. This also implies that instances for different variants of *Flip* do not overlap. So here too it is possible to specify, for every type argument, whether the type has a *Functor* over that variable or not.

A disadvantage of this solution is that every value to which we want to apply a method from the class instance for its type needs to be wrapped with *Flip*, and unwrapped with an *unFlip* call. For example, we could apply a function to a wrapped type as:

$$\mathit{fmap}\ (+1)\ (\mathit{Value}\ 42) \Rightarrow \mathit{unFlip}\ (\mathit{fmap}\ (+1)\ (\mathit{Flip}\ (\mathit{Value}\ 42)))$$

This solution meets our main goal, but the extra boilerplate code necessary to wrap and unwrap datatypes before and after applying type class methods means that it does not satisfy the second condition.

2.4 Associated Type Families

The reason why the *Functor* class needs instances with a variable $f :: * \to *$ is to make it possible to construct $f\ a$ and $f\ b$. It is not vital for *Functor* that f is a

type constructor and a the last argument, the only part that matters is that $f\ a$ is a type that contains a somewhere, and $f\ b$ the same type but with a replaced by b. However, it is currently only possible to declare a *Functor* instance using the type parameter in the last position.

With the `TypeFamilies` language extension of GHC [2] it is possible to define type families within type classes. We can use such an associated type family instead of using $f\ a$ and $f\ b$ directly. This way, the type family indicates how the types are changed within the class functions:

```
class Functor f where
  type FunctorApp f c :: *
  fmap :: (a → b) → FunctorApp f a
                  → FunctorApp f b
instance Functor (Maybe x) where
  type FunctorApp (Maybe x) y = Maybe y
  fmap :: (a → b) → FunctorApp (Maybe x) a
                  → FunctorApp (Maybe x) b
  fmap _ Nothing = Nothing
  fmap f (Just x) = Just (f x)
data Value v u d = Value v
instance Functor (Value v u d) where
  type FunctorApp (Value v u d) v' = Value v' u d
  fmap :: (a → b) → FunctorApp (Value v u d) a
                  → FunctorApp (Value v u d) b
  fmap f (Value v) = Value (f x)
```

Note that *Functor* no longer receives a type of kind $* \to *$ but of kind $*$, because it doesn't apply it: it uses the *FunctorApp* definition for that instead. The type family would have one argument for the type of the instance and one argument for every variable the type class uses. The rhs of the type family should be the first type, with the arguments substituted at the right positions. While this may seem like a lot of extra code for all instances and classes, it is possible to translate definitions written with the current syntax to this format automatically. Only for instances where the extra expressiveness is needed the type families would need to be added by hand.

This solution has a problem due to the way type families currently work: to call *fmap* (+1) on *Just* 1, the type family equality constraint *FunctorApp f a* $\sim$ *Maybe Int* needs to be solved (a substitution for f needs to be found). However, type families in GHC are not injective. There could be other types T which define *FunctorApp T v* = *Maybe x*, so the equality constraint cannot be solved. This makes type families unusable for our goal.

2.5 Conclusion

None of the mentioned solutions satisfies the main goal and the conditions in Sect. 1.

A number of the approaches allow multiple instances per type constructor for a given type class. For example, some allow defining *Functor* instances (or a new instance intended to look like *Functor*, such as *Functor2*) for both the last and the second to last type variable. Although this may be useful, it cannot satisfy both conditions at the same time: without either type annotations or boilerplate code, the compiler is unable to determine which instance to use for an *fmap* call, for example:

$$\begin{array}{l}\textbf{data}\ T\ x\ y\ z = T\ x\ y\\ \textbf{instance}\ \mathit{Functor}\ (\Lambda x\ .\ T\ x\ y\ z)\ \textbf{where}\\ \quad \mathit{fmap}\ f\ (T\ x\ y) = T\ (f\ x)\ y\\ \textbf{instance}\ \mathit{Functor}\ (\Lambda y\ .\ T\ x\ y\ z)\ \textbf{where}\\ \quad \mathit{fmap}\ f\ (T\ x\ y) = T\ x\ (f\ y)\\ \hline \texttt{*Main>}\ \mathit{fmap}\ (+1)\ (T\ 2\ 3)\end{array}$$

We shall therefore consider multiple instances for the same type overlapping, even when they use a different ordering of type variables.

3 Type-Level Lambdas

In the previous sections we have informally used Λ to denote a type-level lambda function in class instance heads. In this section we will give a precise definition of a restricted form of type-level lambdas, and we will investigate the issues it can create with type checking.

3.1 Undecidability

Let us examine more closely why the solution in Sect. 2.1 is rejected by the compiler. Type synonyms can have zero or more arguments. However, contrary to type families they cannot do any case distinction on those variables. We can consider type synonyms polymorphic "functions" on types.

To be able to use a type class function, the compiler needs to be able to determine for each instance of that class whether the type inferred for the function can be made equal to the type of the instance, possibly by substituting some free type variables. If a single instance matches then the class constraint has been resolved and the correct implementation of the class function can be looked up.

Determining whether a substitution of free variables exists which makes a partially applied type synonym equal to a type comes down to determining

whether two lambda expressions are equivalent. This problem is known as unification. Specifically, it is higher-order unification: free variables may be replaced by new lambda abstractions.

Higher-order unification is undecidable in general [6]. Haskell therefore uses the rule that type synonyms must be fully applied before they may be used as a type. That is why the example in Sect. 2.1 is rejected: it is using a partially applied type synonym where a type is expected.

3.2 Adding Type-Level Lambdas

To define an instance of *Functor* where *fmap* has type $(a \to b) \to \mathit{Value}\ a\ u\ d \to \mathit{Value}\ b\ u\ d$, the instance head would need to have a type τ such that $\tau\ a$ is equal to $\mathit{Value}\ a\ u\ d$ and $\tau\ b$ is equal to $\mathit{Value}\ b\ u\ d$. Type synonyms can not be used in instance heads, but even if they could, it would be more convenient to have a notation that does not require defining new type synonyms for every type class and type constructor. Therefore we introduce as new notation the *type-level lambda*: $\Lambda v\ .\ \mathit{Value}\ v\ u\ d$. This is a type where v is bound by the lambda, and u and d are free.

Evaluation is, just like value-level λ-functions, a β-reduction step where the argument is substituted for a variable:

$$(\Lambda x\ .\ M)\ y \to_\beta M[x := y]$$

The kind checking rule for type-level lambdas is given by:

$$\frac{\mathcal{Q}; Q; \Gamma, x : \kappa \vdash T : \kappa'}{\mathcal{Q}; Q; \Gamma \vdash \Lambda x.T : \kappa \to \kappa'}$$

here $\mathcal{Q}$ is a top-level environment and Q is a set of constraints. Γ is a kinding environment, T is a type and κ and κ' are kinds.

In dependently typed languages type-level lambdas and value-level lambdas are the same concept, but also in other, non-dependently typed functional languages the concept exists, for example in Scala, see Sect. 8.2. Although type-level lambdas do not exist in Haskell itself, they do occur in Core, the typed internal representation of GHC.

Note that $\forall\ u\ .\ \mathit{Value}\ v\ u\ d$ and $\Lambda u\ .\ \mathit{Value}\ v\ u\ d$ do not mean the same thing. $\forall\ u\ .\ \mathit{Value}\ v\ u\ d$ has the same kind as $\mathit{Value}\ v\ u\ d$, but $\Lambda u\ .\ \mathit{Value}\ v\ u\ d$ has kind $l \to k$, with $u :: l$ and $\mathit{Value}\ v\ u\ d :: k$.

3.3 Decidable Unification of Type-Level Lambda Terms

We can view a type-level lambda as an anonymous type synonym, just like a value-level lambda function is an anonymous function. Unification of type-level lambdas will therefore be undecidable in general. We can use multiple solutions for this:

1. Use the same restriction as for type synonyms: a type-level lambda needs to be fully applied before it may be used as a type, with an exception for instance heads.
   A disadvantage of this solution is that for example monad transformers, which take a type variable with a *Monad* constraint, cannot be specified for monads that use type-level lambdas. For example, $MaybeT\ (\Lambda u\ .\ Value\ v\ u\ d)\ a$ would be forbidden.

   $$\textbf{newtype}\ (Monad\ m) \Rightarrow MaybeT\ m\ a = MaybeT\ \{runMaybeT :: m\ (Maybe\ a)\}$$

2. We can try to restrict the unification problems to a subset that is decidable. First-order unification (unification where no new λ-abstractions may be introduced) is decidable, but not sufficient for our goal: it would be unable to unify $f\ a$ with anything.
   Higher-order matching is also decidable. Matching is unification where one of the two arguments contains no free variables. However, this is also not useful in Haskell: the type in a type class instances may contain free variables.
3. Use Guided Higher-Order Unification (Λ_{GHOU}) as proposed by [9]. See Sect. 8.1.
4. The above three solutions are either too restrictive or very complicated. Instead, we choose to allow unapplied type-level lambda functions. Except for in instance heads (see Sect. 5.2), type-level lambdas are not unified: they must be α-equivalent or type checking will fail. This implies that they may be used in monad transformers, which is impossible in the first option, but they must be used consistently.

4 Type Checking Haskell

This section briefly describes type checking and constraint solving for Haskell as used in GHC [12].

Type checking is split into two phases: first types are inferred, for which constraints are generated, and then these constraints are solved. Solving these constraints gives a set of substitutions and a (possibly empty) set of unsolved constraints.

4.1 Example

Consider the following program:

$$f\ x = x + 5$$

The type inferencer starts with giving f a function type, because it takes an argument. x gets assigned the argument's type.

$$f :: \alpha \to \beta$$
$$x :: \alpha$$

By looking up the *Num* class, the type inferencer determines that $(+) :: (Num\ a) \Rightarrow a \to a \to a$ and $5 :: (Num\ b) \Rightarrow b$. By looking at the body of f and how it uses $(+)$, the type inferencer introduces the constraints $\alpha \sim a$, $b \sim a$ and $\beta \sim a$. So the type inferencer finishes with the set of constraints $(\alpha \sim a, b \sim a, \beta \sim a, Num\ a, Num\ b)$.

The constraint solver turns these equality constraints into substitutions, as they are all simple. The result is the substitution $[\alpha \mapsto a, \beta \mapsto a, b \mapsto a]$. Due to the substitution, the constraint $Num\ b$ has become unnecessary, as it is equal to $Num\ a$, so only one constraint is left, $Num\ a$. This constraint is left over, which means it gets added to f's type signature, making the final type signature $(Num\ a) \Rightarrow a \to a$.

4.2 Generating Constraints

The important constraints to consider are:

1. **Class constraints:** a class followed by zero or more types:

   $$D\ x$$

   We only look at classes that use exactly one variable, see also Sect. 7.1.
2. **Equality constraints:** constraints requiring two types to be equal:

   $$a \sim b$$

   We consider different types of equality constraints:
   (a) **Impossible:** Equality constraints that contain different concrete types on both sides cannot be solved:

   $$Char \sim Int$$

   This also includes equality constraints where an applied type is matched with an non-decomposable type:

   $$f\ a \sim Int$$

   (b) **Simple:** Equality constraints that contain a single type variable on one side:

   $$a \sim T$$

(c) **Type family:** Type families generate equality constraints that might need to be specified by the programmer manually:

$$F\ a\ \sim\ T$$

(d) **Applied:** Equality constraints that have an applied type variable on one side, and a different applied type variable or a concrete type on the other side:

$$f\ a\ \sim\ g\ b$$
$$f\ a\ \sim\ [Int]$$

4.3 Solving Constraints

To solve the generated constraints, a number of different solvers are applied one after another in a loop. If during one iteration of the loop no changes are made to the set of remaining constraints, the loop terminates and the set of constraints that are left over is returned. If this set is non-empty, then these are usually turned into errors.

The different solvers include:

1. **Canonicalization:** Before constraints are passed to other solvers, they are canonicalized. This makes the constraints simpler and ensures that constraints are always following certain rules. For example, it rejects equality constraints where the same variable occurs on the lhs and the rhs and the lhs and the rhs are not equal, as these would require an infinite type (the "occurs check").
   If a type family is used within another type family, then these are split into two separate type family constraints.
   We use `[W]` to denote wanted constraints (generated during type checking) and `[G]` to denote given constraints (given by the user by supplying a type signature). Some examples of the canonicalization step are:

   $$\{\texttt{[W]}\ f\ a\ \sim\ g\ b\} \rightarrow \{\texttt{[W]}\ f\ \sim\ g, \texttt{[W]}\ a\ \sim\ b\}$$
   $$\{\texttt{[W]}\ f\ a\ \sim\ [Int]\} \rightarrow \{\texttt{[W]}\ f\ \sim\ [\,], \texttt{[W]}\ a\ \sim\ Int\}$$

   This is allowed because f and g have to be (partially applied) type constructors, not type synonyms or type families. As described in Sect. 1, every type constructor has at most one partially applied type for a given kind so it can be unambiguously resolved.
2. **Binary interaction:** Another solver looks at two canonical constraints together. For example, a simple equality constraint and another constraint will apply the equality constraint as a substitution to the other constraint.

Having two identical constraints implies one of them can be deleted. Type family constraints with identical lhs, but different rhs generate an equality constraint between the rhs and allow one of the two type family constraints to be deleted. The binary interaction rules only look at two constraints that are either both given, or both wanted.
Here are some examples of the binary interaction step:

$$\{\texttt{[W]}\ \mathit{Num}\ a, \texttt{[W]}\ \mathit{Num}\ a\} \rightarrow \{\texttt{[W]}\ \mathit{Num}\ a\}$$
$$\{\texttt{[W]}\ a \sim T, \texttt{[W]}\ \mathit{Num}\ a\} \rightarrow \{\texttt{[W]}\ a \sim T, \texttt{[W]}\ \mathit{Num}\ T\}$$
$$\{\texttt{[W]}\ F\ \mathit{Int} \sim [a], \texttt{[W]}\ F\ \mathit{Int} \sim [\mathit{Int}]\} \rightarrow \{\texttt{[W]}\ [a] \sim [\mathit{Int}], \texttt{[W]}\ F\ \mathit{Int} \sim [\mathit{Int}]\}$$

3. **Simplification:** The simplifier also looks at two canonical constraints, but specifically pairs of constraints where one of them is given and the other is wanted.
Obviously, a given constraint and a wanted constraint that are identical implies that the wanted constraint can be deleted. Given simple equality constraints are be used as substitutions on wanted constraints.
Here are some examples:

$$\{\texttt{[W]}\ \mathit{Functor}\ f, \texttt{[G]}\ \mathit{Functor}\ f\} \rightarrow \{\texttt{[G]}\ \mathit{Functor}\ f\}$$
$$\{\texttt{[W]}\ \mathit{Functor}\ f, \texttt{[G]}\ f \sim g\} \rightarrow \{\texttt{[W]}\ \mathit{Functor}\ g, \texttt{[G]}\ f \sim g\}$$

4. **Top-level interaction:** The top-level interaction stage is the stage where the class instances, type family instances and equality constraints given in the code are used to solve wanted constraints. During top-level interaction, the instances of type classes and type families are used to solve wanted type class and type family constraints, respectively. This may introduce new constraints, for example for superclasses of instances.
For example, if the usual *Functor* [ ] instance is in scope, then we may eliminate all *Functor* [ ] constraint:

$$\{\texttt{[W]}\ \mathit{Functor}\ [\,]\} \rightarrow \{\,\}$$

Suppose we have a type family:

$$\textbf{type family}\quad F\ x\quad ::\ *$$
$$\textbf{type instance}\ F\ \mathit{Int} = [\mathit{Int}]$$

Then the top-level interaction step produces the following constraint:

$$\{\texttt{[W]}\ a \sim F\ \mathit{Int}\} \rightarrow \{\texttt{[W]}\ a \sim [\mathit{Int}]\}$$

5 Adding Type-Level Lambdas to GHC

We have started to include our proposed changes in GHC. Our development started off with the 7.7 version of GHC, which is the development branch that was later released as GHC 7.8.

The changes to GHC 7.7 consist of three parts: the parser is modified to allow the notation `/\` for type-level lambdas, the internal representation of types is modified to support Λ and the constraint solving is adapted to take the possibility of type-level lambdas in class instance heads into account.

`/\` is currently valid syntax for a term-level operator. Because our extension is type-level syntax this does not cause problems. With the GHC extension `TypeOperators` [11], `/\` can be used as a valid type operator. We assume that because using it requires a relatively uncommon extension, there will not be many problems with existing code already using this syntax.

5.1 Parser

The changes to the parser are simple: `/\` follows the same rules as `forall`: `/\` must be followed by one or more types, which can optionally have a kind signature when the `KindSignatures` extension [11] is enabled. For example:

```
/\ a . [a]
/\ x . ()
/\ (f :: * -> *) . f Int
```

5.2 Evaluation of Type-Level Lambdas

We evaluate type-level lambdas greedily during type checking: when a type-level lambda is encountered that is applied to an argument, the substitution is carried out immediately. We show that this cannot introduce non-termination in the type checker.

The type-level language of Haskell can be considered a "typed" lambda calculus, where Haskell's kinds form the types. The kinds form a simply-typed lambda calculus, thus the types are strongly normalizing. This implies that we cannot write non-terminating combinators from the untyped lambda calculus, such as Ω:

$$\omega = (\lambda x.x\ x)$$
$$\Omega = \omega\ \omega$$

A value-level ω combinator cannot be type checked as its type fails the occurs check: suppose $\omega :: \tau \to \sigma$, then $\sigma = \tau\ \tau$, which implies $\tau = \tau \to \sigma$. This equation cannot hold for types. Thus Ω also fails to type check. In GHC, trying to define ω would cause a "occurs check" error. At the type-level, defining ω (by either a type-level lambda or a type synonym) is also rejected, as the occurs check for its kind would fail.

Another way we can achieve non-termination is through recursion. Haskell **let**-bindings can refer to themselves, which can lead to infinite recursion. It is impossible to create a recursive definition from lambda functions alone: they are anonymous, so cannot refer to themselves. It would be possible if the Y combinator were available, but a type-level Y combinator is impossible for the same reason as Ω: it fails the kind occurs check, in the same way that the Y combinator, without using newtypes, fails the type occurs check in Haskell.

But, just like how **let**-bindings in Haskell allow terms to be named, we can give names to type-level lambdas by defining a type-level function. Haskell currently supports two different forms of type-level functions:

- Type synonyms
- Type families

Type synonyms are not allowed to be recursive: trying to create a (mutual) recursive type synonym will give an error "Cycle in type synonym declarations".

Type families can be (mutually) recursive, but only when the `UndecidableInstances` [11] extension is enabled. When this extension is not enabled, recursion is forbidden because it will imply that an equation has a rhs that does not follow the rules which require the rhs to be "smaller" than the type family arguments.

Turning on `UndecidableInstances` causes GHC to lift many of its restrictions that guarantee termination. With this flag on, it is already possible to create infinite loops in the type checker, using only type families. Adding type-level lambdas does not cause code that was previously terminating to be come non-terminating.

Type-Level Lambdas in Instances. The main goal of our work is to allow type-level lambdas in the heads of type class instance declarations. To avoid problems with undecidability here, we will only allow well-formed type-level lambdas as instance heads.

Definition 1. *A* well-formed *type-level lambda function is an expression that can be constructed using the following grammar:*

$$
\begin{array}{rr}
T & \textit{(type constructor)} \\
a & \textit{(type variable)} \\
as := a_1, \ldots, a_n & \textit{(1 or more)} \\
ts := t_0, \ldots, t_m & \textit{(0 or more)} \\
t := a \mid T\ ts & \textit{(monotype)} \\
L := \Lambda as.T\ t\ s & \textit{(type lambda)}
\end{array}
$$

under the extra condition that every variable that is bound by a `/\` *must occur exactly once as an argument to the inner type constructor. In other words, a type-level lambda is well-formed if every lambda bound type variable is used exactly once, and the body of the lambda is either again a type-level lambda, or starts with a type constructor.*

Some examples of well-formed types:

$$\Lambda x\ .\ \mathit{Value}\ v\ u\ d$$
$$\Lambda x\ .\ [x]$$
$$\Lambda x\ .\Lambda y\ .\Lambda z\ .\ \mathit{Value}\ z\ x\ y$$

and some not well-formed types:

$$\Lambda x\ .\ \mathit{Value}\ [x]\ y\ z$$
$$\Lambda x\ .\ \mathit{Value}\ v\ x\ z$$
$$\Lambda x\ .\ [\mathit{Int}]$$

The advantage of only allowing well-formed type-level lambdas is that their unification is simple, which avoids the undecidability involved with higher-order unification.

For most instances given by programmers the well-formedness restriction should not cause problems: it allows reordering of type variables (which is what we want to do), but no more complicated type-level functions. In this sense reordered instances are just as powerful as instances which can be written without this extension.

We give some examples of instances that cannot be written because they use non well-formed type-level lambdas.

- Definition 1 states that the body of a type-level lambda starts with either another type-level lambda, or a type constructor. In particular, it does not start with a type variable, including the lambda-bound type variables. This implies for example that an instance for $\Lambda x\ .\ x\ \mathit{Int}$ cannot be specified. However, this type would have kind $(* \to *) \to *$ (which is not the same as $* \to * \to *$). Classes using type variables of this kind, or even higher kinds, are quite rare, at least for now.
- A type-level lambda bound type variable occurs as a direct argument to the inner type constructor. It may not be inside another type constructor in the argument. This implies that, for example, $\Lambda x\ .\ \mathit{Maybe}\ [x]$ is not well-formed. We believe this will not be a problem for Haskell programmers, as type class arguments are currently always interpreted to refer to one of the direct arguments of a type constructor.
- A type-level lambda bound type variable occurs exactly once as an argument to the inner type constructor. For example, $\mathit{Functor}\ (\Lambda x\ .\ (x, x))$ is not well-formed. Just like the previous case, we believe this will not be a problem for Haskell programmers, as type class arguments are currently always interpreted to refer to exactly one of the direct arguments of a type constructor, never multiple at the same time.

– From a category theoretic point of view it might be interesting to define the identity functor and functor composition. We can express these with type-level lambdas as:

```
instance Functor (Λx . x) where
  fmap f x = f x
instance (Functor f, Functor g)
  ⇒ Functor (Λx . f (g x)) where
  fmap f x = fmap (fmap f) x
```

However, this does not work, as both instances' heads are not well-formed. Although these instances are interesting, they are not very useful in Haskell. First of all, with the way instances are resolved in Haskell the identity functor would overlap with every other possible *Functor* instance. When the user would call *fmap f* with the intention to use the identity functor (i.e., on a type with no other functor instance), *f x* can do the same. Secondly, when using *fmap* on composed functors, GHC cannot resolve whether the call to *fmap f* is meant to apply on the first functor alone, or on the composition. *fmap* (*fmap f*) can be used instead to apply to the composition.
Alternatively, it is possible to use the wrappers *Identity* and *Compose* from the `transformers` package [4] to obtain identity functors and functor composition.

5.3 Constraint Solving

We can now express type class instances with type-level lambda instances, but to use them the constraint solver needs to find and use those instances. This does not, however, require many changes to the class constraint solver. The changes are mostly in the solving of equality constraints, specifically applied equality constraints.

Firstly, the splitting of applied equality constraints as happens during canonicalization (Sect. 4.3) is no longer allowed. Suppose we have the following datatype, and in the code *fmap* is applied to it:

```
data T x y z = T x y z
foo = ...fmap g (T 1 () 'a') ...
```

During type inference, the constraint $f\ a\ \sim\ T\ Int\ ()\ Char$ will be created. However, we do not yet know which *Functor* instance for T exists. The possible decompositions are therefore:

– $f\ \sim\ \Lambda b\ .\ T\ b\ y\ z$ and $a\ \sim\ Int$
– $f\ \sim\ \Lambda b\ .\ Value\ v\ b\ z$ and $a\ \sim\ ()$
– $f\ \sim\ \Lambda b\ .\ Value\ v\ y\ b$ and $a\ \sim\ Char$

The constraint cannot be split, but it has to be solved as a whole. First, we give the requirements for how these constraints can be split (an applied type variable

on one side, a concrete type on the other) and second we explain how to deal with the case where both sides consist of an applied type variable.

Applied-Concrete. Suppose a constraint $f\ a\ \sim\ \mathit{Value}\ v\ u\ d$ is encountered: an applied type variable on one side, with a concrete type on the other side.

As mentioned in Sect. 2.5, satisfying all the conditions from Sect. 1 at once can only be done when every type allows only one instance per type class. We shall therefore keep this restriction and use it to solve these constraints. When an applied equality constraint is encountered with a concrete type on the rhs, and a type class is known that applies to that concrete type, then we use the type-level lambda used in the class instance to pick the correct decomposition of the equality constraint.

For example, suppose the following constraints are given:

- The wanted equality constraint: $f\ a\ \sim\ \mathit{Value}\ v\ u\ d$,
- for a certain class C, the class constraint $C\ f$ is given or wanted,
- and exactly one instance of C for one of the types $\Lambda q\ .\ \mathit{Value}\ q\ y\ z$, $\Lambda q\ .\ \mathit{Value}\ v\ q\ z$ or $\Lambda q\ .\ \mathit{Value}\ v\ y\ q$

then we may conclude that $f\ \sim \Lambda ?\ .\ \mathit{Value}\ v\ u\ d$ and thus $a\ \sim ?$ (where ? is determined by the alternative chosen in the third condition). This may sound restrictive, but in many cases all three will be true. If the first two conditions hold, but the third does not, constraint solving fails due to a missing instance of C for T anyway.

A situation where the first condition holds, but no C exists for the second and the third conditions should be rare, however, not impossible. When $f\ a$ is given in a type signature this almost certainly means there is also a constraint on f: without a constraint, it is impossible to obtain a value of type a or $f\ b$ from a value of type $f\ a$, so the function can only produce values of type $f\ a$. This implies the function has a more general type: by replacing $f\ a$ by a new type variable. For example, it is possible to write:

$$\mathit{const} :: f\ a \to b \to f\ a$$
$$\mathit{const}\ x\ \_ = x$$

This function can only be applied to datatypes with at least one argument, but it does not use that information. It could be replaced by:

$$\mathit{const} :: c \to b \to c$$
$$\mathit{const}\ x\ \_ = x$$

We do not expect this restriction to have impact on existing Haskell code.

Applied-Applied. For an equality constraint of the following form:

$$f\ a\ \sim\ g\ b$$

we use a similar rule as in Sect. 5.3: if we have a class constraint that applies to both f and g, then we may split the constraint into $f \sim g$ and $a \sim b$.

When we cannot find any constraint on both f and g, then the constraint stays unsolved. Maybe a different equality constraint can find a substitution for f or g and the constraint will be solved later. If that does not happen, then this constraint is reported to the user as an error. There is one exception to this rule:

$$f\ a \sim f\ b$$

is decomposed automatically, resulting only in $a \sim b$.

Type Rules. We express the typing rules from the previous two sections formally in Fig. 3. Using the notation from In [12], $\hookrightarrow$ is a typing judgment with an input tuple $\langle \bar{\alpha}, \varphi, Q_g, Q_w \rangle$ and an output tuple $\langle \bar{\alpha}', \varphi', Q_g', Q_w' \rangle$. Here, $\mathcal{Q}$ is the top-level environment (containing, for example, all defined class instances), Q_w are the wanted constraints and Q_g are the given constraints. $\bar{\alpha}$ is a set of touchable variables (the variables which may be substituted) and φ is a set of substitutions. The $\hookrightarrow$ judgment is applied until a fixed-point is found. In [12] a number of cases for $\hookrightarrow$ are described, Fig. 3 adds a new case to deal with type-level lambdas in instances. We have removed a case that is not explicitly given in [12], namely the rule that automatically decomposes $f\ a \sim g\ b$ into $f \sim g$ and $a \sim b$.

The first part of the decompose function in Fig. 3 checks the top-level environment for an instance of the class C for the type f, then for an applied-concrete equality constraint and a constraint $C\ f$, which may have come from either the given or wanted constraints. The equality constraint is then decomposed according to the type-level lambda used by the instance. The second part of the decompose function checks for an applied-applied equality constraint and a type class that applies to both. Then the equality constraint is split. Here $C\ f$ and $C\ g$ may also come from both the wanted and the given constraints.

$$\begin{aligned}
&\text{decompose}(\mathcal{Q} \wedge C\ (\Lambda \overline{y_i}.T\overline{y}), f\ \overline{a} \sim T\overline{x}, C\ f) \\
&\qquad = (f \sim \Lambda x_i.T\overline{x}) \wedge \overline{(a \sim x_i)} \\
&\text{decompose}(\mathcal{Q}, f\ \overline{a} \sim g\ \overline{b}, C\ f \wedge C\ g) \\
&\qquad = (f \sim g) \wedge \overline{(a \sim b)}
\end{aligned}$$

$$\frac{\text{decompose}(\mathcal{Q}, Q_1, Q_2 \wedge Q_3) = Q_4}{\begin{array}{c}\mathcal{Q} \vdash \langle \bar{\alpha}, \varphi, Q_g \wedge Q_3, Q_w \wedge Q_1 \wedge Q_2 \rangle \hookrightarrow \\ \langle \bar{\alpha}, \varphi, Q_g \wedge Q_3, Q_w \wedge Q_2 \wedge Q_4 \rangle\end{array}}$$

Fig. 3. The applied-concrete and applied-applied type checking rules

5.4 Termination

The goal of the type checker in Haskell is to judge whether programs are well-behaved within a finite number of steps. It is not a problem if programs are rejected when they cannot be determined to be well-behaved, but the type checker should not accept a not well-behaved program. In particular, this implies that it is important that the constraint solver terminates. To any type, we can assign a *depth* as follows:

- The depth of a single type variable or a nullary type constructor is 0.
- The depth of an applied type $t\ a\ b\ c...$ is:

$$1 + \max(\text{depth}(t), \text{depth}(a), \text{depth}(b), \text{depth}(c), \dots)$$

When an applied equality constraint is solved according to one of the two rules we introduced, the result is a number of new equality constraints. These can again be applied equality constraints. For example:

$$[[a]] \sim f\ (g\ x)$$
$$[a] \sim g\ x, [] \sim f$$
$$a \sim x, [] \sim g, [] \sim f$$

However, the newly introduced equality constraints always have a strictly lower depth than the equality constraint that is solved, because solving a constraint cannot introduce a deeper or equally deep nesting level. This implies that infinite loops in equality constraints are impossible: an equality constraint can only introduce a finite number of equality constraints of lower depth.

5.5 Implementation

The described changes were implemented as a patch for GHC. The patch works with the development version 7.7 and can be found on https://github.com/xnyhps/ghc/tree/TypeLambdaClasses.

6 Results

The code shown in Sect. 1 now works: *fmap* changes the values of the first type argument of the datatype. This example shows that no extra type annotations are necessary, and the *Functor* class used is unchanged. This implies that the solution satisfies the two conditions from Sect. 1.

Here is an example of a *Monad* instance, again defined on the first type argument of a datatype with three arguments. Additionally a *MaybeT* monad transformer is used where the same type-level lambda is used. The returned result shows that the correct instance is found.

```
import Control.Monad.Trans.Class
import Control.Monad.Trans.Maybe
```

```
data Value v u d = Value v
instance Monad (Λv . Value v u d) where
  return v = Value v
  (>>=) (Value v) g = g v
bar :: Value String Char Int
bar = return "bar"
foo :: MaybeT (Λv . Value v Char Int) String
foo = lift bar

---

*Main> runMaybeT foo
Value (Just "bar")
```

7 Compatibility with Other GHC Features

To become part of GHC, our changes should not only be consistent with the Haskell 2010 specification [8], but we should also make sure they work correctly with other GHC extensions, or, if that is impossible, document why combining those extensions leads to problems and add warnings to the compiler when users try to use them at the same time.

7.1 MultiParamTypeClasses

The `MultiParamTypeClasses` GHC extension [11] allows type classes to be specified with multiple type parameters. Combining multiple type parameters with type-level lambda instances can create new ambiguities:

```
class C f a b where
  func :: f a b
instance C (Λx y . (x, y)) Int Char where
  func = (1, 'a')
instance C (Λx y . (y, x)) Char Int where
  func = (2, 'b')
```

Without β-evaluating the instance's type-level lambdas, it is not clear that these instances overlap. However, *func* in both instances has the type $(Int, Char)$.

Ambiguity is not necessarily a problem: GHC does not prohibit two instances to exist that can be ambiguous for some type, only when a class is resolved and multiple instances match an error is raised. However, ambiguity complicates the solver's strategy. The solver currently looks for instances for every possible lambda with the required number of arguments for a single type. When using multiparameter type classes, it needs to look for every possible lambda abstraction for each of them. This means that increasing the number of parameters

can lead to an exponential increase in the number of instances that need to be considered.

Instead of failing on ambiguity only when it occurs, another option is to apply the following restriction to multiparameter type-level lambda instances: different instances with the same type constructor on the lambdas body must use the same type-level lambda. This forbids the given example, because while both use the type constructor $(,)$, the order in which they take their arguments is flipped. The advantage of this restriction is that the type-level lambda for every parameter can be found independently, avoiding the exponential increase in cases.

7.2 PolyKinds

Combining the `PolyKinds` extension [11] with type-level lambda instances currently has some implementation problems. However, we expect that with some more work it is possible to eliminate those problems.

When `PolyKinds` is enabled, type constructors take implicit kind arguments for all type variables which do not have a fixed kind. For example, the constructor *Value* of the datatype:

$$\textbf{data}\ \mathit{Value}\ v\ u\ d = \mathit{Value}\ v$$

does no longer have type $\mathit{Value} :: \forall\ u\ d\ .\ v \to \mathit{Value}\ v\ u\ d$, but instead:

$$\begin{aligned}\mathit{Value} :: \forall\ &(k :: \square)\\ &(l\ :: \square)\\ &(v :: *)\\ &(u :: k)\\ &(d :: l)\ .\\ &v \to \mathit{Value}\ k\ l\ v\ u\ d\end{aligned}$$

In practice the user does not see these kinds, so when decomposing an applied equality constraint involving *Value*, these extra kind arguments should not be considered when selecting the type-level lambda to use. However, they currently are, causing unsolved constraints in certain situations.

8 Related Work

8.1 Guided Higher-Order Unification

As already mentioned in Sects. 3.3 and 5.2, [9] introduce a similar approach to type-level lambdas in instance declarations. They introduce a more involved unification strategy known as Guided Higher-Order Unification (Λ_{GHOU}). This strategy also keeps unification decidable, but in a less restrictive way than our approach. To maintain decidability, Λ_{GHOU} applies the restrictions to solving type-level Λ constraints:

- Identity type-level functions are not allowed.
  $\Lambda x\ .\ x$ is forbidden.
- Constant type-level functions are not allowed.
  This forbids, for example $\Lambda x\ .\ Int$.
- Type-level projection functions are not allowed.
  This forbids functions like $\Lambda x\ y\ .\ x$.

Additionally, given an instance:

instance $C\ (\Lambda x_1 \ldots x_n . T\ t_1 \ldots t_m)$ **where**

they apply the following restrictions:

- Each variable that is free in $\Lambda x_1 \ldots x_n . T\ t_1 \ldots t_n$ occurs once.
- Each x_i occurs free in $T\ t_1 \ldots t_n$.
- Each t_i is either one of the x_j, or equal to $g\ y_1\ y_l$, where $g \notin \{x_i\}$ and $\{y_j\} \subseteq \{x_i\}$.

The well-formedness restriction we introduce is more strict. We discuss the differences. Firstly, Λ_{GHOU} allows lambda-abstracted variables to occur multiple times in the lambda's body. Our well-formedness restriction forbids this and only allows those variables to occur exactly once. So these are valid Λ_{GHOU} types, but not well-formed:

$\Lambda x\ .\ Value\ x\ x\ y$
$\Lambda x\ .\ (x, x)$

Secondly, Λ_{GHOU} allows lambda-abstracted variables to occur arbitrarily "deep" within the lambda's body, in other words, nested in (an) extra type constructor(s). Thus, these are also valid Λ_{GHOU}, but not well-formed:

$\Lambda x\ .\ Value\ [x]\ y\ z$
$\Lambda x\ .\ [(x, Int)]$

Λ_{GHOU} applies the same restriction on overlapping instances as our approach: instances using different Λ-abstractions, but the same class and type, overlap. This implies that an instance for $[(x, Int)]$ will still overlap all other instances using $[a]$.

As we argued in Sect. 5.2, we believe the extra limitations we impose on type-level lambdas in type classes will not be a problem for many programmers, as allowing these would create instances that are very different from how type classes are currently used. Programmers can create instances where the order of type variables doesn't matter, which was our initial goal.

8.2 Scala

Scala allows type-level lambdas using the following syntax:

```
({type ?[a] = Either[A, a]})#?
```

Here, `{type ?[a] = ... }` introduces a new type alias named `?` with a single argument `a`. It is defined to be equal to `Either[A, a]`. Finally, `#?` projects the `?` type from the type alias.

This allows class instances to be defined as in Fig. 4.

Scala also allows multiple, different instances for the same type. As mentioned in Sect. 2.5, this requires a trade-off: in Scala this is done by giving instances names and passing these instances to the function with a class constraint. This is more manual work for the programmer, but it implies that the implementation of the type checker can be much simpler than the approach described here: the type checker always knows which instance to use before type checking starts; it does not need to find the instance based on the types in the constraints.

```
trait Monad[M[_]] {
  def point[A](a: A): M[A]
  def bind[A, B](m: M[A])(f: A => M[B]): M[B]
}

class EitherMonad[A]
    extends Monad[({type ?[a] = Either[A, a]})#?] {
  def point[B](b: B): Either[A, B]
  def bind[B, C] (m: Either[A, B])
                 (f: B => Either[A, C]): Either[A, C]
}
```

Fig. 4. The *Monad* trait (the Scala equivalent of a type class) in Scala, with an instance for *Either* using a type-level lambda, where the second type variable is used by the *Monad*

9 Conclusion

We have described a problem caused by the inflexibility of the combination of type classes and type constructors with multiple arguments. We have presented a number of potential solutions to this problem that are currently supported by GHC and explained how none of them satisfies a number of desired properties.

We introduce a restricted form of type-level lambdas in Haskell, which we have added to a development version of GHC. The restrictions are necessary to avoid undecidability in the type checker. Our solution is simpler than previous attempts, the main task is to adapt the constraint-solver to deal with our form of type-level lambdas. Despite its simplicity, our approach is powerful enough to solve the inflexible type class instances problem in a way that satisfies the desired properties. The solution has virtually no impact on existing Haskell code.

Acknowledgments. We would like to thank the anonymous reviewers for their extensive comments.

References

1. Alkemade, T.P.: UnitTyped (2012). https://hackage.haskell.org/package/unittyped. Accessed 9 Apr 2014
2. Chakravarty, M.M.T., Keller, G., Jones, S.P., Marlow, S.: Associated types with class. In: Proceedings of POPL 2005: The 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 1–13. ACM (2005)
3. Elliott, C.: TypeCompose (2007–2013). https://hackage.haskell.org/package/TypeCompose. Accessed 10 Dec 2013
4. Gill, A., Paterson, R.: Transformers (2009–2012). Accessed 10 Dec 2013
5. Hall, C., Hammond, K., Jones, S.P., Wadler, P.: Type classes in haskell. ACM Trans. Program. Lang. Syst. **18**, 241–256 (1996)
6. Huet, G.P.: The undecidability of unification in third order logic. Inf. Control **22**(3), 257–267 (1973)
7. Kmett, E.A.: Bifunctors (2011–2013). https://hackage.haskell.org/package/bifunctors. Accessed 10 Dec 2013
8. Marlow, S.: Haskell 2010 language report (2010). https://www.haskell.org/onlinereport/haskell2010/
9. Neubauer, M., Thiemann, P.: Type classes with more higher-order polymorphism. In: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming, ICFP 2002, pp. 179–190. ACM, New York (2002)
10. Peterson, J., Jones, M.: Implementing type classes. In: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, PLDI 1993, pp. 227–236. ACM, New York (1993)
11. The GHC Team: The Glorious Glasgow Haskell Compilation System User's Guide. http://www.haskell.org/ghc/docs/7.6.3/html/users_guide/index.html
12. Vytiniotis, D., Jones, S.P., Schrijvers, T., Sulzmann, M.: OutsideIn(X) Modular type inference with local assumptions. J. Funct. Program. **21**(4–5), 333–412 (2011)

Laminar Data Flow: On the Role of Slicing in Functional Data-Flow Programming

Research Paper

Baltasar Trancón y Widemann[1,2(✉)] and Markus Lepper[2]

[1] Technische Universität Ilmenau, Ilmenau, Germany
baltasar.trancon@tu-ilmenau.de
[2] Semantics GmbH, Berlin, Germany

Abstract. We use the concept of laminar flow, as opposed to turbulent flow, as a metaphor for the decomposition of well-behaved purely functional data-flow programs into largely independent parts, necessitated by aspects with different execution constraints. In the context of the total functional data-flow language Sig, we identify three distinct but methodologically related implementation challenges, namely multi-rate scheduling, declarative initialization, and conditional execution, and demonstrate how they can be solved orthogonally, by decomposition using the standard program transformation technique, slicing.

1 Introduction

In fluid dynamics, *laminar flow* is an ideal transport process where a fluid flows in essentially parallel layers (lamina) at *different speeds*, without *turbulent* interference. We borrow the term as a useful metaphor for the benevolent properties of purely functional data-flow programs, in particular their decomposeability. Form this perspective, we investigate the various uses of decomposing a data-flow network into lamina, in the context of an effective language implementation. All our uses are concerned with lamina delimited by a functional *aspect*, that is, a data-flow closure of variables of interest. As such, they are instances of the well-known program transformation technique *slicing* [17].

This semiformal presentation retains a high level of abstraction from technical details throughout, in order to make the conceptual uniformity and naturality of the approach stand out, and the overarching story concise and readable. The issues of formally precise definition, soundness, completeness and complexity of methods are out of scope here, to be discussed in technical companion papers.

The structure of this paper and its novel contributions are organized as follows: Sect. 2 introduces the basic semantic design of the Sig language, as defined in previous technical papers [14]. Section 3 specifies the treatment of multi-rate systems in Sig, so far characterized largely by example [16]. Section 3.1 gives a novel local scheduling algorithm for Sig multi-rate systems. Section 4 summarizes the Sig approach to delayed computations from [14], and then gives a semiformal but complete specification of the novel extension to non-literal initial value

M. Serrano and J. Hage (Eds.): TFP 2015, LNCS 9547, pp. 85–103, 2016.
DOI: 10.1007/978-3-319-39110-6_5

expressions. Section 5 motivates and discusses a transformation to conditionally executed code, so far published only in German as a Master's Thesis [9].

2 The Sig Language

The Sig language is a novel, total, purely functional data-flow programming language [13,14]. Its aim is to allow the expression of synchronous data stream processing algorithms in an elegant declarative style, with semantics clean and simple enough for domain experts without professional computer science background to experience programming as an orderly mathematical activity rather than an exercise in some 'black art' of tinkering and hacking.

Central to the semantics of Sig is a compositional view on synchronous data-flow computations, no matter whether primitive operations, subclauses, or complex networks, as clocked Mealy machines with private state. In [14] we have specified the formal semantics framework and a stack of program transformations that normalize higher-level functional programs into machines, compare also [8]. State spaces are inferred from the use of quasi-functional *delay* operators; the programmer never manipulates state variables directly.

Data flow is synchronous in a strict sense: all values are communicated as if by shared memory, where many readers and a single writer are arbitrated by an external clock. Race conditions, messages, events and other observable side effects are forbidden by the semantics. The following additional features of Sig semantics are of particular interest for the present paper, as they each give rise to a different application of program slicing:

1. The writer and reader of a variable are implicitly synchronized, that is, operating at the same rate; the writer always before the reader for a given clock tick. The exceptions are *up-* and *downsampling* connectors which transmit data between subnetworks operating at distinct, interleaved clock rates.
2. A delay operator applied to a data stream prepends one (or more) values to the stream. These initial values may be defined by complex expressions, with the possibility to share work between the initialization and running phase of a network, and the obligation to check for causality violations by initial values depending on-line input.
3. Sig expressions are totally functional, that is, they produce no side effects during their evaluation (*pure*), and they may neither diverge nor abort nor block (*total*). Thus control flow can be implemented transparently, either as conditional evaluation of alternative subexpressions, or as 'posthumous' selection of alternative subresults, whichever is more convenient. The transformation of functional front-end programs into the machine representation naturally yields the latter, but many sequential execution platforms favor the former.

The following sections explore these applications in turn, in the same order as they occur in the Sig compiler pipeline.

Since the actual front-end notation of Sig is irrelevant for the present discussion, we present example programs in pseudocode.

$$\left[\frac{x \to s}{s := 0\,;\,(s + x)}\right] \qquad \left[\frac{x \to s}{s := (0\,;\,s) + x}\right]$$

Fig. 1. Simple Sig component definitions

Components are the first-class citizens of the language. They can be thought of as stream-processing functions, but in contrast to lambda expressions, inputs and outputs are notated symmetrically; both are named and can occur in any multiplicity. We enclose component definitions in brackets, enumerate inputs and outputs without giving their types, separated by a line from a block of assignments that constitutes the component body, and use mathematical operators and constants with their obvious semiformal meaning. For instance, the example components depicted in Fig. 1 each have a single input x and output s, respectively, and operate on some unspecified numerical data type.

We write $i\,;s$ for the initialized single-step delay operator with initial element i prepended to delayed stream s (sometimes written *fby* or $\gg$ in other data-flow languages). That is, if the expression s attains the stream of values s_n for $n = 0, 1, \ldots$, then $t = i\,;\,s$ attains values t_n where $t_0 = i$ and $t_{n+1} = s_n$. For instance, the two examples in Fig. 1 each define a component that outputs a cumulative sum s of its input stream x.

In the left component, the sum is initially zero, and each input element is added by the next clock tick. Hence this version has a latency of one tick, whereas the right component is latency-free; each input element contributes to the sum immediately. That is, $s_n = \sum_{k<n} x_k$, or $s_n = \sum_{k\leq n} x_k$, respectively. Note the pattern of delayed feedback that is ubiquitous in synchronous data-flow algorithms; other forms of recursion are forbidden in Sig. Component bodies are understood as systems of equations; evaluation order is implicitly constrained by data-flow dependencies only.

3 Slicing for Multi-Rate Data Flow

Algorithms implemented in Sig are on-line side-effect-free computations on data streams. Streams are accessed in a very disciplined way: there is no random access, only the element associated with the current clock tick is available, and if past elements are needed they must be retained explicitly, using delay. In summary, data flow behaves as if each conceptual stream is realized as a single clocked buffer variable.

While this model of communication is rather restrictive, it is very easy to grasp and use correctly and reliably, and there are various different application domains where algorithmic requirements fit this pattern neatly. In particular, Sig algorithms can run in real time, given sufficient computational resources, because they never violate causality or productivity: values may never depend on the future or circularly on the present, nor take infinitely many steps to compute. We have demonstrated the use of the Sig language and its Java-based execution system in moderate real-time settings by creating a simple but non-trivial, polyphonic live audio synthesis system with interactive MIDI input [15].

The audio domain has a characteristic feature shared by many other real-time application fields: subsystems operate at various rates, with the slow parts controlling parameters of the fast parts (*modulation*), and the fast parts in turn providing summary information to the slow parts for feedback (*aggregation*). For instance in audio parlance,

- *wave generators* operate at *audio rate*, such as 44 kHz (CD), 96 kHz (studio);
- *modulators* of parameters such as pitch, volume and filter shape operate at *control rate*, defined as either a fraction, such as 1/64, of audio rate, or as a fixed rate, such as 1 kHz;
- *notes* and other *sequencer events* are controlled at a yet much slower rate, such as the MIDI resolution of 24 per quarter note, or the infamous 120 bpm 'techno' beat;
- some computations concern only *initialization*, and operate at rate zero.

Data-flow networks written in Sig are not declared explicitly to operate at particular rates. Rather, they constrain usable rates implicitly at two different levels of abstraction:

- The abstract program itself imposes a system of *qualitative* constraints, that is, equations and inequations between the rates of inputs and outputs, by its formal *synchronization* properties.
- Any concrete implementation imposes additional *quantitative* constraints, that is, ranges of achievable rates, by its technical *throughput* limits.

Since the latter can only be discussed properly in very detailed technological context, we focus here on the former, which can be understood in terms of a few language primitives and a static analysis.

The default behavior of primitive operations in Sig is to synchronize their in- and outputs. Assume that a program has been reduced to a core representation in static single-assignment (SSA) form, as discussed in detail in [14]. Then any assignment of the form

$$y_1, \ldots, y_n := f(x_1, \ldots, x_m)$$

that is, any primitive hyperedge of the data flow graph, induces equations on the rates of all concerned variables:

$$R(x_1) = \cdots = R(x_m) = R(y_1) = \cdots = R(y_n) \tag{1}$$

It follows for a whole network, that variables are synchronized if and only if they are connected by a path; data-independent subsystems can run at independent rates. However, this does not yet allow for interference such as modulation or aggregation. To this end, we add *directed resampling* primitives to the language. The operation

$$y := \mathrm{upsample}(x)$$

is a functional identity, such that $x = y$ holds instantaneously at all times, but y is allowed to be (re)used at a faster rate than x is produced.

$$R(x) \leq R(y) \tag{2}$$

Note that data flow from slow to fast subsystems is taken as instantaneous: whenever clock ticks for x and y coincide, y reflects the new value of x immediately. This also allows us to subsume constant values consistently as data streams of rate zero, provided that all clocks begin to tick simultaneously at initialization time of the system ('big bang').

By contrast, the converse operation

$$y := \mathrm{downsample}(i, x)$$

allows y to be used at a slower rate than x is produced. Namely, at each clock tick of y, the most recent value of x is observed. Whenever clock ticks for x and y coincide, y reflects the *previous* value of x. Obviously, the value of y at initialization time is not determined by x; this is the purpose of the additional, type-compatible input i; compare the analogous, synchronized expression $i \,;\, x$.

$$R(y) \leq R(x) \tag{3}$$

The asymmetry of these two operations ensures that the scheduling strategy of subsystems at different rates is independent of the actual program: the instantaneous data flow at each coincidence of ticks is always from slow to fast.

Note that, when components are composed, their rate constraints accumulate. Since all inequalities are non-strict, there composite system is always satisfiable. However, it is possible for rates to be equated emergently in the composite, say, by an inequation $R(x) \leq R(y)$ arising from one component, and a converse inequation $R(y) \leq R(x)$ independently from another component. Resampling operations along the path are rendered ineffective; the rate analysis pass should notify the programmer about this potential usage error.

As an example of rate analysis and separation, consider the following manifestly multi-rate program:

$$\left[\begin{array}{ll} () \to \mathit{wave}, \mathit{high} & \\ \hline \mathit{wave} := \mathrm{upsample}(\mathit{amp}) \cdot \sin(\mathit{phase}) & m := \max(\mathrm{abs}(\mathit{wave}), (0 \,;\, m)) \\ \mathit{phase} := 0 \,;\, (\mathit{phase} + \alpha) & \mathit{high} := \mathrm{downsample}(0, m) \\ \mathit{amp} := 1 \,;\, (\mathit{amp} \cdot \gamma) & \end{array}\right]$$

It produces an oscillation *wave* with current phase *phase* and amplitude *amp*, which increase arithmetically and geometrically with parameters α and γ, respectively. Since amplitude is a long-term modulating property in relation to phase, the former is upsampled. Additionally the attained maximum absolute value m of *wave* is recorded, and a downsampled copy *high* provided for monitoring.

Note that this program is not in proper SSA form, because there remain nested expressions with unnamed intermediate variables. However, these play no significant role in the rate analysis of the program. We shall take the same liberty for harmless abbreviation in the following examples.

Note also the references to α and γ, which are parameters of the generic component definition, and become private life-time constants for each component

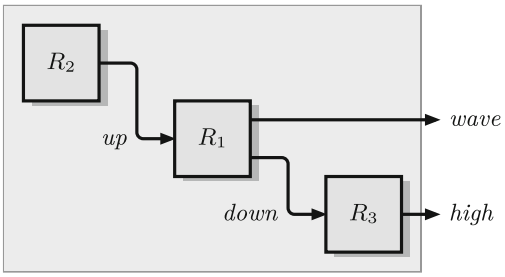

Fig. 2. Single-rate slices of example multi-rate component

instance. They are supplied by the higher-order programming mechanism of Sig, the details of which are out of scope here.

For the example, straightforward application of rules (1)–(3) finds a synchronous cluster $R_1 R(wave) = R(phase) = R(m)$, and both $R_2 = R(amp) \leq R_1$ and $R_3 = R(high) \leq R_1$. Thus for instance, setting R_1 to audio rate, R_2 to control rate, and R_3 to the refresh rate of a graphical output device would yield a consistent real-time execution context.

If different subnetworks are to be actually operated at different rates, they can no longer be implemented directly as the transition of a monolithic Mealy machine. Rather, the component should be sliced according to synchronicity, and each slice translated to machine form independently. To this end, each resampling operator is split into a fresh input–output pair of matching variables s^+, s^-, respectively, and the program is sliced backwards, based on the synchronicity partition of both original and synthetic outputs.

For our example, we obtain two synthetic variable pairs, $up^\pm$ and $down^\pm$, where $R(up^-) = R_2$ and $R(down^-) = R_1$, and thus the following three subcomponents:

$$\left[\begin{array}{l} up^+ \to wave, down^- \\ \hline wave := up^+ \cdot \sin(phase) \\ m := \max(\mathrm{abs}(wave), (0\,;m)) \\ phase := 0\,;(phase + \alpha) \\ down^- := 0\,;m \end{array}\right] \qquad \left[\begin{array}{l} () \to up^- \\ \hline amp := 1\,;(amp \cdot \gamma) \\ up^- := amp \end{array}\right] \qquad \left[\begin{array}{l} down^+ \to high \\ \hline high := down^+ \end{array}\right]$$

The runtime scheduler takes care of the 'anionic' asynchronous data flow $up^- \rightsquigarrow up^+$ and $down^- \rightsquigarrow down^+$ behind the scenes; see Fig. 2. Downsampling is translated to delay at the operand rate. Note that the third component is trivial and serves only to mask the synthetic variable $down^+$ behind the original variable *high*; by contrast, the second component exposes the previously internal original variable *amp* as the synthetic output up^-.

3.1 Transparent Local Scheduling

Under certain mild assumptions, the slices of a component for different rates can be reassembled, as a component operating at the fastest concerned rate. The slower rates are triggered intermittently by an internal, local scheduler, whose action is transparent to the component's environment. Local scheduling may or may not be an option, depending on the technical context and the actual rate proportions; however, the mere possibility adds to the compositionality of the language, and hence merits some consideration. Furthermore, it is also interesting from a purely algorithmic perspective.

Assume that a component operates at a finite number of distinct rates, $0 < R_1 < \cdots < R_n$. Assume furthermore that these rates are *commensurable*, that is in rational proportion: there are integer numbers $0 < \rho_1 < \cdots < \rho_n$ such that $R_i = \rho_i R_0$ for some *fundamental* rate R_0, which need not be present in the component.

Equivalently, let $0 < T_n < \cdots < T_1$ denote the periods of operation, with $T_i = R_i^{-1}$. Then there are integer numbers $0 < \delta_n < \cdots < \delta_1$ such that $T_i = \delta_i T_*$ for some *atomic* period T_*; namely $\delta_i = \rho_*/\rho_i$ and $T_* = T_0/\rho_*$, where ρ_* is the least common multiple of $\{\rho_1, \ldots, \rho_n\}$.

By normalization, we obtain rational numbers $0 < \pi_1 < \cdots < \pi_n = 1$, where $\pi_i = \rho_i/\rho_n = \delta_n/\delta_i$. The time of the k-th tick of the i-th clock is given as $t_{i,k} = k \cdot \delta_i$.

Now assume that it is valid to quantize all clocks at the fastest occurring rate ρ_n, as opposed to the atomic rate ρ_*, as long as the causal order is maintained. When a tick of the (slower) i-th clock falls *between* two ticks of the (fastest) n-th clock, say $t_{n,m} < t_{i,k} < t_{n,(m+1)}$, it can be safely quantized to the *successor*, since the operational model of Sig explicitly allows instantaneous data flow from slow to fast subcomponents. From the perspective of the n-th clock, we obtain the most recent tick of the i-th clock by rounding $t_{n,k}$ first *down* to a multiple of δ_i, then we obtain its quantization by rounding *up* to a multiple of δ_n. This is conveniently expressed as $q_{i,k} = \delta_n \lceil \lfloor k \cdot \pi_i \rfloor / \pi_i \rceil$.

Note that, with respect to the exact tick sequence t_i, q_i is both rounded up to a multiple of δ_n and stretched out to match the pacing of $t_n = q_n$; whereas t_i is injective, q_i has runs of average length π_i^{-1}. Note also that $q_{i,k} \leq q_{n,k}$. Morally, the i-th component is still assumed to operate at a constant rate, reflected by the special nullary Sig primitive **dt**, which evaluates to its own clock period and is a lifetime constant for each component instance. Unless a component is run in an embedded context and connected to very hard real-time input/output, the micro-latency induced by quantification goes undetected.

The local scheduler, which is operated at the fastest rate R_n, needs to perform a computation of the i-th subcomponent at its k-th invocation, if and only if $q_{i,k} = q_{n,k}$. Conveniently enough, this can be achieved by a variant of Bresenham's algorithm for quantized line drawing [1]. Scheduling a commensurable two-rate component for one period of its fundamental rate is analogous to drawing a rastered two-dimensional straight line with extent $\Delta x = \rho_2$ and $\Delta y = \rho_1$: advancement by one pixel in the x and y dimension corresponds to quantized clock ticks at ρ_2 and ρ_1, respectively.

```
var s := 0
invariant −ρ2 < s ≤ +ρ1
for each step do
  let up := s ≥ 0
  s := s + ρ1
  if up then
    s := s − ρ2
    step component 1
  end if
  step component 2
end for
```

Fig. 3. Component scheduling, Bresenham style

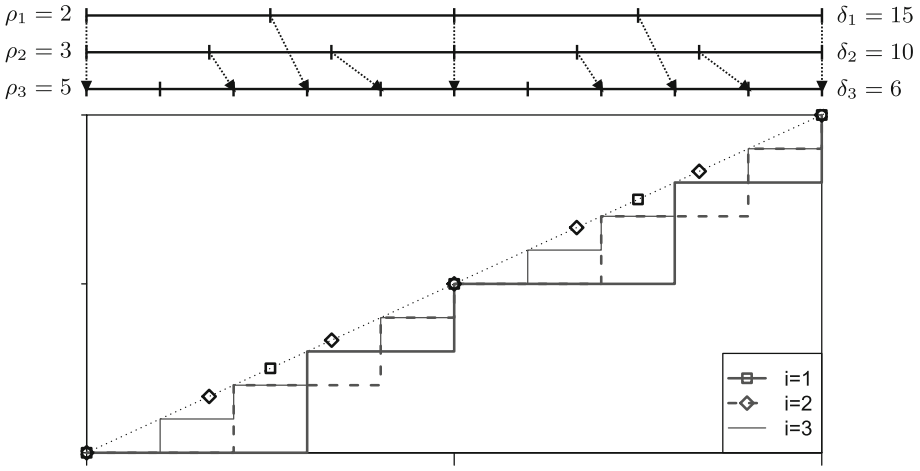

Fig. 4. *Top* – quantization timelines; $n = 3$, $\rho_* = 30$, interval of $2T_0$ shown. *Bottom* – discrete (clock) time over continuous (real) time; tick sequences t_i as points on identity line (*dashed*), quantized time q_i as step functions.

Note that we have $0 < \Delta y \leq \Delta x$, the base case of Bresenham's algorithm to which other cases are reduced. Thus at most one tick at rate R_1 happens per tick at rate R_2. The algorithm is adapted by changing the rounding mode, and omitting the main loop such that one turn is performed at each invocation of the component, and the slope is extrapolated indefinitely to the right. Figure 3 shows the basic algorithm in pseudocode.

For $n > 2$ components, additional counters $s_2, \ldots, s_{n-1}$ can be added. For rates R_i that also are multiples of, or may be quantized to, R_j for some $j < n$, the scheduling problem can be decomposed hierarchically. The latter approach is generally more efficient, in particular when $R_j \ll R_n$.

Figure 4 shows an example multi-rate ensemble with relative rates $\rho = (2, 3, 5)$, depicting the evolution of its three clocks over an interval of $2T_0$.

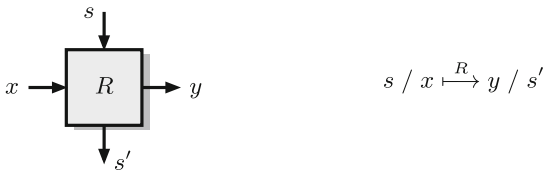

Fig. 5. Diagram depiction and syntax for state transitions.

4 Slicing for Declarative Initialization

From the semantics perspective it is tempting to neglect the initial values of delayed streams as a minor detail. Indeed, the pattern seen in the preceding examples, namely delayed feedback to a monoid operation, with the monoid unit as the natural initial value, is quite common, and suggests an implicit solution by inference. Note that some clocked synchronous data-flow formalisms omit the specification of initial values altogether, for instance Faust [10]. However, not all common uses of delay fit the bill; an example is given in (†) on p. 10 below, after a summary of the Sig implementation of delay [14]. In this section, we discuss a slicing technique to address the issue.

Sig front-end programs are neutral about whether each name is bound to a single value or a stream. For the domain of synchronous data-flow algorithms, this is an elegant and adequate abstraction: virtually all primitive computations operate element-wise anyway, such that the distinction would provide no insight; the only, but ubiquitous exception being delay operations.

In the semantics as specified in [14], delay operations are replaced by private (buffer) state, by a syntax-directed program transformation. The resulting SSA-style intermediate representation can be read directly as element-wise formal semantics, namely as the transition rule of a Mealy machine, in the form of a quaternary relation $R \subseteq (S \times A) \times (B \times S)$, where A, B are the products of ranges of input (x) and output (y) variables, respectively, and S is the product of ranges of inferred state variables, in the double role of pre-state (s) and post-state (s'); see Fig. 5.

In this view, a single-step delay operation is simply the special case of a square identity $\delta_A = I_{A \times A} \subseteq (A \times A) \times (A \times A)$: at each clock tick, the pre-state becomes output, while the input simultaneously becomes post-state, to be output in the next cycle, etcetera.

These element-wise transition relations can be depicted graphically and admit three different meaningful compositions, namely *parallel* ($\|$), *functional* ($\circ$) and *temporal* ($\fatsemi$) composition, respectively; see Fig. 6. The desired stream-wise semantics of a data-flow program is 'morally' the infinite temporal replication of the corresponding element-wise transition relation:

$$\lim_{n \to \infty} \underbrace{R \fatsemi \cdots \fatsemi R}_{n}$$

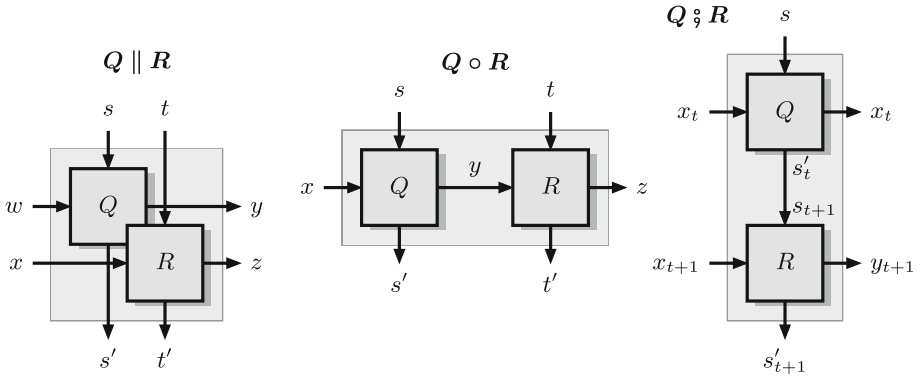

Fig. 6. Composition axes of transition relations, adapted from [14]

It takes an initial pre-state and a whole input stream to a whole output stream; there is no final post-state. Between clock ticks, post-state is fed back to pre-state. In [14] we have given a rigorous coinductive construction.

However, there is a catch: because the initial states are outside of this semantic interpretation, they can only be given as uninterpreted constants. In practice, one would certainly like to have the full expression language to denote complex initial values. Hence one-off initialization and repeated element-wise computation should be compiled together, and only separated for code generation by static analysis and slicing. For example, consider the following program

$$\left[\begin{array}{l} () \to x \\ \hline x := 1\,;y \\ y := (a\,/\,2)\,;(a \cdot y - x) \\ a := 2 - \alpha \cdot \alpha \end{array}\right] \qquad (\dagger)$$

which computes a very resource-efficient, numerically stable approximation of the sequence $x_n = \cos(n \cdot \alpha)$. It uses a magic constant a both at initialization and at each clock tick. The remainder of this section specifies a general program analysis and transformation in several steps, interleaved with applications to this example for immediate illustration.

The embedding of initialization expressions works as follows: For each variable v that is the result of a delay operation,

$$v := i\,;u$$

perform a *statification* (sic): introduce a pre–post pair of fresh matching state variables s_v, s'_v, respectively; then add a pair of statements according to the semantics of delay given earlier in this section

$$v := s_v \qquad\qquad s'_v := \iota(i,u)$$

where ι is a special primitive, a variant of the well-known ϕ operator of SSA, which selects its first operand when evaluated during initialization of the component, and its second operand otherwise. For the example, we obtain:

$$\left[\begin{array}{ll} s_x, s_y \;/\; () \to x \;/\; s'_x, s'_y & \\ \hline x := s_x & s'_x := \iota(1, y) \\ y := s_y & s'_y := \iota(a \;/\; 2, a \cdot y - x) \\ a := 2 - \$step \cdot \$step & \end{array}\right]$$

Note the slash notation to enclose the input/output interface in the state context, as in Fig. 5.

The subsequent static analysis works as follows. Several 'virtual' slices are are formed based on forward or backward data flow:

- The forward slice D (*dynamic*) for all statements depending on pre-state and/or input;
- the backward slice I (*initial*) for all statements affecting post-state, considering only the first operand of each ι operator;
- the backward slice L_0 (*loop*) for all statements affecting output and/or post-state, considering only the second operand of each ι operator; this slice is split into the subslice L of statements also contained in D plus their *immediate* data-flow predecessors, and its relative complement $\ell = L_0 \cap \overline{L}$. The intuition here is that statements in L/ℓ are directly/indirectly relevant for the loop phase, respectively.

They yield a multidimensional classification: statements...

- in $\overline{I} \cap \overline{L_0}$ are dead (they play no role for output or state);
- in $I \cap \overline{L_0}$ are computed for initialization only (they play no role for loop output or post-state);
- in $D \cap I$ are causally illegal attempts to read from a stream during initialization (they depend on loop input or pre-state but affect initial state), except for the safe case of ι operations whose first operand is not in D;
- in $D \cap L$ are recomputed at each clock tick (they depend on loop input or pre-state and affect output or post-state);
- in $\overline{D} \cap L$ are loop invariant, computed at initialization and retained as constants (they do not depend on loop input or pre-state but directly affect output or post-state);
- in ℓ are computed privately at initialization, and used by the preceding (they do not depend on loop input or pre-state but indirectly affect output or post-state).

The full classification of a statement consists of two binary decisions, namely $\{D, \overline{D}\}$ and $\{I, \overline{I}\}$, and a ternary decision, $\{\overline{L_0}, L.\ell\}$. The classification of a statement is inherited by its result variable(s).

For the example, we find that $x, y \in D \cap \overline{I} \cap L$ and $s'_x, s'_y \in D \cap I \cap L$ and $a \in \overline{D} \cap I \cap L$ (and its unnamed intermediates in $\overline{D} \cap I \cap \ell$). It follows that a needs to be retained as a constant.

This is achieved by a transformation that introduces synthetic delay with identical feedback: replace each statement of the form

$$c := e$$

where e is in $\overline{D} \cap L$, with

$$c := e \,;\, c$$

and apply ι-introduction as above. For the example, we obtain:

$$\left[\begin{array}{ll} \multicolumn{2}{l}{s_x, s_y, s_a \;/\; () \to x \;/\; s'_x, s'_y, s'_a} \\ \hline x := s_x & s'_x := \iota(1, y) \\ y := s_y & s'_y := \iota(a \;/\; 2, a \cdot y - x) \\ a := s_a & s'_a := \iota(2 - \alpha \cdot \alpha, a) \end{array}\right]$$

The ι-introduction rule has created trivial copy statements. Clean up by performing copy propagation on statified variables, substituting s_v for v, with one crucial exception: for references to statified variables v in the *first* operand of a ι operator, s'_v is substituted instead.

For the example, we obtain:

$$\left[\begin{array}{ll} \multicolumn{2}{l}{s_x, s_y, s_a \;/\; () \to x \;/\; s'_x, s'_y, s'_a} \\ \hline x := s_x & s'_x := \iota(1, s_y) \\ & s'_y := \iota(s'_a \;/\; 2, s_a \cdot s_y - s_x) \\ & s_a := \iota(2 - \alpha \cdot \alpha, s_a) \end{array}\right]$$

In the final step, two slices are computed:

- an *initialization* slice, which retains just the statements affecting post-state, and replaces each ι operator by its first operand;
- a *loop* slice, which retains all statements affecting post-state and/or output, and replaces each ι operator by its second operand.

For the example, we obtain:

$$\left[\begin{array}{l} s_x, s_y, s_a \;/\; () \to x \;/\; s'_x, s'_y, s'_a \\ \hline s'_x := 1 \\ s'_y := s'_a \;/\; 2 \\ s'_a := 2 - \alpha \cdot \alpha \\ \hdashline x := s_x \\ s'_x := s_y \\ s'_y := s_a \cdot s_y - s_x \\ s'_a := s_a \end{array}\right]$$

where initialization and loop are above and below the dashed line, respectively.

The execution model assumes that the initialization slice is evaluated once; afterwards and for conceptually infinitely many clock ticks, post-state is fed back

to pre-state and the loop slice is evaluated. Note that, at least for common simple cases, suitable efficient implementations of state feedback can be suggested by peephole optimizations: for the example,

- the statement $s'_a := s_a$ witnesses that a is constant, and can be eliminated in the obvious way by allocating s_a and s'_a to the same storage location;
- the statement $s'_x := s_y$ witnesses that the pair y, x is a buffer queue. While this particular instance is of trivial size and needs no special attention, longer queues that are candidates for a ring buffer implementation can be found by simple flow graph pattern matching.

5 Slicing for Conditional Execution

Sig has no concept of true user-defined control flow, such as jumps, loops or recursion; the infinite unfolding of output streams is the only means of iteration. However, the language does have *pattern matching* constructs as expressive and general means of dynamic case distinction. Because all Sig expressions are totally productive (may not diverge element-wise), case distinction can be seen as a data-flow, rather than control-flow issue: the semantics allows for all alternative rules to be evaluated concurrently. Such speculative evaluation may *fail* on a matching rule because of a refuted pattern on the left hand side. In that case, the right hand side is taken to evaluate to the special value $\perp$, which can be conceived of as an exception.

In the core language, a special primitive γ is used to guard the actual result of the right hand side, conditional on the success of matching. From the set of alternatives, a second special primitive φ selects a successful rule, conceptually catching the exceptions. All other operations are strict with respect to $\perp$. Our usage of φ differs from its classical namesake ϕ in SSA in the sense that choice is not based on incoming control flow; instead it is generally nondeterministic but avoids $\perp$ whenever possible. If rules are mutually exclusive and jointly total, as they are in a normalized well-defined pattern-matching expression, then the final result is total and deterministic. A static analysis enforces that $\perp$ is never leaked from a component.

As in the previous section, we interleave general descriptions of compilation tasks, and particular illustrations. For example, consider the following Sig program, which defines a wave generator component with an additional switch to silence the output, a *gate* in audio parlance:

$$\left[\begin{array}{l} gate \rightarrow wave \\ \hline wave := \textbf{if } gate \textbf{ then } \sin(phase) \textbf{ else } 0 \\ phase := 0\,;\,(phase + \alpha) \end{array}\right]$$

The if-then-else construct is syntactic sugar for pattern matching on the enumerated datatype $\{\text{true}, \text{false}\}$. Pattern matching is eliminated according to a nontrivial syntax-directed program transformation duly specified in [14], and

replaced by a network of φ and γ operators. For the example, performing also statification of delay as described in the previous section, we obtain:

$$\left[\begin{array}{l} s \;/\; gate \to wave \;/\; s' \\ \hline wave := \varphi(a, b) \\ \quad a := \gamma(\sin(phase), gate = \text{true}) \\ \quad b := \gamma(0, gate = \text{false}) \\ phase := s \qquad\qquad\qquad\qquad s' := \iota(0, phase + \alpha) \end{array}\right]$$

The φ operator selects the output value from either of the complementary branches a and b. Each of these is given by a γ operation that yields the first operand if the constraint expressed by the other operand(s) is satisfied, or $\bot$ otherwise. In a well-typed context, exactly one branch is defined (non-$\bot$) at all times. For the example, by copy propagation and initialization slicing as described in the previous section, we obtain:

$$\left[\begin{array}{l} s \;/\; gate \to wave \;/\; s' \\ \hline s' := 0 \\ \hdashline wave := \varphi(a, b) \\ \quad a := \gamma(\sin(s), gate = \text{true}) \\ \quad b := \gamma(0, gate = \text{false}) \\ \quad s' := s + \alpha \end{array}\right]$$

On massively parallel execution platforms, such as FPGAs, it is perfectly reasonable and efficient to evaluate both branches independently and implement φ as a multiplexer. From this perspective, the whole body of a component is just a single basic block. By contrast, on more conventional sequential platforms, such as ordinary CPUs and virtual machines, it is often desirable to save time by evaluating only relevant branches. To this end, statements should be organized in smaller basic blocks guarded by conditional branching instructions, as usual for conventional sequential imperative languages.

The Sig compiler, which currently targets the Java virtual machine platform, has an experimental pass for automatic sequentialization of programs with φ and γ operations, thus converting data flow into control flow. It works roughly as follows [9]:

- For each variable, determine the condition under which it is defined. Since the only sources of undefinedness are failed pattern-matching operations, this is a straightforward backwards data-flow analysis. The result is a positive propositional formula, where pattern matching primitives, γ and φ contribute literals, conjunctions and disjunctions, respectively.
- Group statements according to the definition conditions of their results.
- Nest groups according to logical implication of their conditions. Simplify nested conditions relative to their parents.
- Split each group into as few basic blocks as possible, such that inter-group data-flow dependencies do not connect blocks circularly.

– Guard the entry into each basic block by conditional branching.
– Re-interpret φ operations as their traditional SSA ϕ counterparts.
– Optionally, follow up with a standard SSA cleanup pass, which attempts to allocate all operands of a ϕ operation to the same storage location, thus eliminating it completely; otherwise, the set of ϕ operations at the beginning of a basic block is transposed to a set of simultaneous moves at the end of each predecessor block [5].

Note that it may appear simpler to sequentialize case distinctions directly as they appear in the front-end syntax. But code transformations such as common subexpression elimination and algebraic simplification are dramatically effective on the pure data-flow form, and can disrupt the original block structure.

For the example, on the whole we obtain a sequential program that can be described by the following pseudocode with conditional execution:

$$\left[\begin{array}{l} s \;/\; gate \rightarrow wave \;/\; s' \\ \hline s' := 0 \\ wave := \sin(s) \quad \textbf{if } gate = \text{true} \\ wave := 0 \quad\;\; \textbf{if } gate = \text{false} \\ \quad s' := s + \alpha \end{array}\right]$$

Note that both a and b have been re-allocated to *wave*, and all administrative moves have been eliminated thereby.

It follows that the, possibly expensive, expression $\sin(s)$ is only evaluated when necessary. However, this example is misleading about the generality of the approach as outlined above: it depends crucially on all conditional statements being stateless primitives; if calls to dynamically bound, potentially stateful subcomponents are allowed, as they are in full Sig, then naïve conditional execution is no longer safe.

Consider a variation of the preceding example, where the waveform shape is no longer a pure function, but a statically non-fixed stream generator component reference:

$$\left[\begin{array}{l} gate \rightarrow wave \\ \hline wave := \textbf{if } gate \textbf{ then } shape() \textbf{ else } 0 \end{array}\right]$$

This can be seen as a modularization, where the concerns of wave generation, including private phase state, and of gate operation are separated. The original form is restored by plugging in the following simple generator:

$$\left[\begin{array}{l} () \rightarrow wave \\ \hline wave := \sin(phase) \\ phase := 0\,;\,(phase + \alpha) \end{array}\right]$$

The problem with this 'morally equivalent' situation is that, by design, the component instance state referred to by *shape* is private, and the output and transition operations are tied up together in a monolithic quaternary machine relation, as discussed in the preceding section. Hence the call to the subcomponent cannot be assumed to be side-effect-free, and hence it cannot be simply

omitted at some clock ticks, lest spurious local time-freezes arise. For the example, the phase of the generator would simply stop whenever the gate is shut down, which may or may not be pragmatically acceptable depending on context, but is certainly not in accordance with the principle of least surprise.

A number of partial or universal solutions to this dilemma are conceivable:

- Simply call subcomponents unconditionally, wasting any opportunity for inter-component work saving.
- Record statelessness in the type system; call only manifestly stateless subcomponents conditionally.
- The original version that has no issues can be restored by inlining; simply defer code generation until parameters are bound.
- Generally separate state transition and output production, delegate the former to a central scheduler. However, separation can be difficult because of arbitrary data-flow inter-dependencies, and decentral solutions are decidedly more light-weight and elegant.
- Create an alternative and more efficient, *tacit* variant of each component, to be used under conditions where the outputs are not needed.

The last solution has little impact on our execution model as described before, and good modularity. Furthermore, it can be implemented by straightforward program slicing, namely as the backward slice of all statements affecting post-state, with no outputs. For the simple wave generator, after initialization slicing we obtain the component on the left, and its tacit variant on the right:

$$\left[\begin{array}{l} s \;/\; () \to \mathit{wave} \;/\; s' \\ \hline s' := 0 \\ \mathit{wave} := \sin(s) \\ s' := s + \alpha \end{array}\right] \qquad \left[\begin{array}{l} s \;/\; () \to () \;/\; s' \\ \hline s' := 0 \\ s' := s + \alpha \end{array}\right]$$

Note that, just like in the monolithic original example, the tacit variant saves work by not computing $\sin(s)$. The two component definitions are understood to be instantiated together, and share state.

The sequentialization algorithm can be fixed to support stateful subcomponents by adding the following clause:

- After grouping statements by condition, for each statement that calls a subcomponent c, add a corresponding call to the tacit variant c_0 under the complementary condition.

Note that, since all condition literals are about closed algebraic datatypes, the complement of a condition is again a finite positive formula. The rule is sound but redundant also for unconditional calls; the additional tacit call is unreachable by construction.

For the modularized example, we obtain:

$$\left[\begin{array}{ll} \mathit{gate} \to \mathit{wave} & \\ \hline \mathit{wave} := \mathit{shape}() & \textbf{if } \mathit{gate} = \text{true} \\ () := \mathit{shape}_0() & \textbf{if } \mathit{gate} = \text{false} \\ \mathit{wave} := 0 & \textbf{if } \mathit{gate} = \text{false} \end{array}\right]$$

In general, components may have more than one output variable. In that case, there are candidates for intermediate variants between the full and the tacit version; namely one for each subset of outputs. Each of these variants can be obtained by the same straightforward slicing policy. But because of exponentially growing number of combinations and diminishing relative gains, we do not envisage a static exploration. Nevertheless, intermediate slices may occur dynamically on execution platforms supporting run-time program specialization.

6 Conclusion

6.1 Related Work

The design of Sig draws on inspirations and results from many paradigms. Several characteristic features are inherited from the 'French school' of synchronous languages; in particular we are indebted to Lucid Synchrone [2] for the initialized delay operator and implicit handling of (multi-)rate. However, Lucid suffers from being based on an impure functional core language. Faust [10] has pioneered the virtue of purity and its benefits in powerful code transformations. Functional reactive programming (FRP) based on the theory of arrows [6], provides many of the elegant core properties we aim at, although in a more abstract and general, and less down-to-earth setting. Extensions of its axiomatic theory exists to deal with advanced features such as complex initialization and control flow, namely as *causal commutative arrows* [8] and *arrows with choice* [7], respectively, although a grand unifying picture remains elusive.

There are remarkably few attempts to apply slicing to functional or data-flow languages: Ganapathy and Ramesh apply slicing to a statechart variant [4]. Their definition of a slice is based on behavioral equivalence with respect to one selected output signal and thus quite different from ours. Clarke et al. [3] apply slicing to VHDL, a declarative hardware description language. Astonishingly, they take the detour of converting VHDL into an imperative language (C) and then apply a commercial slicing tool, instead of exploiting the functional aspects of VHDL. A positive example of applying slicing to pure functional programs by pure functional means is due to Rodrigues and Barbosa [12]; while they operate very elegantly on an abstract semantic formalism (Bird–Meertens), they cannot promise that their approach does scale, what can easily be shown for our approach, which is syntax-directed and thus more directly applicable.

For the embedding of explicit case distinctions in functional data-flow formalisms, see [11] and [7] for monads and arrows, respectively.

6.2 Summary and Outlook

Because of the rigorously puristic and mathematically elegant design of the Sig semantics framework, very basic and easily implemented, data-flow-based program analyses and transformations go a long way. We have demonstrated the

wide applicability of slicing transformations to a number of conceptually independent issues: advanced support for high-level front-end features, such as multi-rate networks with resampling; necessary support for orthogonality of basic features, such as complex expressions and initialized delay; optimized support for resource-efficient, state-transparent embedding of modular subcomponents.

The whole Sig compiler pipeline is a prototype and subject to ongoing research and development at various ends. Of the slicing applications we have presented here, initialization slicing is stably implemented in the current version of the compiler. Tacit slicing is implemented in principle, but its practical use is deferred pending the integration of the very recent, externally developed sequentialization pass. Multi-rate slicing is in the process of implementation, and the current main focus of our efforts, since it would allow us to raise our demonstrations in the application domain of audio and digital music [15] to a significantly more sophisticated level. Encouraging preliminary results and a first non-trivial case study are discussed in [16].

References

1. Bresenham, J.E.: Algorithm for computer control of a digital plotter. IBM Syst. J. **4**(1), 25–30 (1965)
2. Caspi, P., Pouzet, M.: Lucid Synchrone, a functional extension of Lustre. Technical report, Université Pierre et Marie Curie, Laboratoire LIP6 (2000)
3. Clarke, E.M., Fujita, M., Rajan, S.P., Reps, T., Shankar, S., Teitelbaum, T.: Program slicing of hardware description languages. Technical report, Carnegie Mellon University (1999)
4. Ganapathy, V., Ramesh, S.: Slicing synchronous reactive programs. SLAP 2002, ENTCS **65**(5), 50–64 (2002). doi:10.1016/S1571-0661(05)80440-2
5. Hack, S.: Register Allocation for Programs in SSA Form. Ph.D. Thesis, Universität Karlsruhe, (2007). http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/6532
6. Hudak, P., Courtney, A., Nilsson, H., Peterson, J.: Arrows, robots, and functional reactive programming. In: Jeuring, J., Jones, S.L.P. (eds.) AFP 2002. LNCS, vol. 2638, pp. 159–187. Springer, Heidelberg (2003). doi:10.1007/978-3-540-44833-4_6
7. Hughes, J.: Programming with arrows. In: Vene, V., Uustalu, T. (eds.) AFP 2004. LNCS, vol. 3622, pp. 73–129. Springer, Heidelberg (2005)
8. Liu, H., Cheng, E., Hudak, P.: Causal commutative arrows. J. Funct. Program. **21**, 467–496 (2011). doi:10.1017/S0956796811000153
9. Loth, A.: Synthese von Kontrollfluss für eine Synchrone Datenflusssprache. Master's thesis, Technische Universität Ilmenau (2015)
10. Orlarey, Y., Fober, D., Letz, S.: Syntactical and semantical aspects of Faust. Soft Comput. **8**(9), 623–632 (2004). doi:10.1007/s00500-004-0388-1
11. Petricek, T., Mycroft, A., Syme, D.: Extending monads with pattern matching. In: Haskell Symposium 2011, pp. 1–12. ACM (2011). doi:10.1145/2034675.2034677
12. Rodrigues, N.F., Barbosa, L.S.: Slicing functional programs by calculation. In: Beyond Program Slicing. Dagstuhl Seminar **05451** (2005)
13. Trancón y Widemann, B., Lepper, M.: tSig: Towards semantics for a functional synchronous signal language. In: KPS 2011, Arbeitsbericht des Instituts für Wirtschaftsinformatik 132, pp. 163–168. Westfälische Wilhelms-Universität Münster (2011). https://www.wi.uni-muenster.de/sites/default/files/publications/arbeitsberichte/ab132.pdf

14. Trancón y Widemann, B., Lepper, M.: Foundations of total functional dataflow programming. In: MSFP 2014, EPTCS 153, pp. 143–167 (2014). doi:10.4204/EPTCS.153.10
15. Trancón y Widemann, B., Lepper, M.: Sound, soundness: practical total functional data-flow programming. In: FARM 2014, pp. 35–36. ACM (2014). Demo abstract. doi:10.1145/2633638.2633644
16. Trancón y Widemann, B., Lepper, M.: The Shepard tone, higher-order multi-rate synchronous data-flow programming in Sig. In: FARM 2015, pp. 6–14. ACM (2015). doi:10.1145/2808083.2808086
17. Weiser, M.: Program slicing. In: ICSE 1981, pp. 439–449. IEEE (1981)

A Shallow Embedded Type Safe Extendable DSL for the Arduino

Pieter Koopman(✉) and Rinus Plasmeijer

Institute for Computing and Information Sciences, Radboud University, Nijmegen, The Netherlands
{pieter,rinus}@cs.ru.nl

Abstract. This paper extends our method to construct a shallow embedded domain specific language, DSL, embedded in a function programming language. We show how one can add functions and tasks that are typed by the type system of the functional host language.

The DSL is clearly separated from its host functional language to facilitate the compilation to small executables in C++. The type system of the host language verifies the types in the DSL, including the types and proper use of variables. The DSL is extendable by new language constructs and interpretations without breaking any existing code. The type system guarantees that everything used in a DSL program is properly defined. We apply these techniques for a DSL to program Arduino microprocessor systems from Clean. The long term goal is to incorporate these microprocessors in the iTask system.

1 Introduction

The internet of things, IoT, will for a large part consist of devices equipped with a small microprocessor executing some tailor made program. The Arduino is a family of popular open-source microcontroller boards [2,3]. The Arduino Uno is the archetype of these development boards. The first version was released in 2005. The current version, R3, of this board contains a 8-bit ATMega328 microprocessor running at 16 MHz. It provides 32 KB of flash memory and 2 KB RAM. This board is very suited for control tasks since it provides 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, and serial communication via an USB connection. Arduino boards can be extended by shields. These shields provide various kinds of additional input/output options like motor controls, Blue Tooth communication, Ethernet, LCD, buttons and relays. These elementary and cheap systems are extremely well suited for simple input/output intensive control tasks.

From a software point of view these tiny systems are too small to run any operating system. The standard way to program Arduino's uses it's own dialect of C++. The Arduino IDE compiles this to binary code. This code is uploaded to the Arduino via an USB connection. For this purpose every Arduino is shipped with a tiny boot loader. Arduino programs define two functions for creating a runnable program. The `setup()` function is executed once to initialize the board,

M. Serrano and J. Hage (Eds.): TFP 2015, LNCS 9547, pp. 104–123, 2016.
DOI: 10.1007/978-3-319-39110-6_6

like the constructor of an object. The `loop`() function is executed repeatedly until the board powers off. Both of these functions can be empty. Since there is no operating system or other thread, executing the `loop`() over and over again is the only thing the microprocessor has to do and can do.

Since the Arduino boards are very well suited to control input/output devices and cheap, it is tempting to use them for interfacing iTask programs with the real world [19]. Due to the limited hardware provided by the Arduino board, it is unfeasible to execute complete high level functional programs with these microprocessors. To prevent that we need two separate programs, one for the Arduino and one iTask program, we introduce a Domain Specific Language[1] to program Arduino's. In the near future we want to incorporate this language in the iTask system. This paper focuses on the architecture of our ARDSL, for **Ar**duino **DSL**. Due to the limited power of the Arduino and the necessity to interact with the world ARDSL has imperative components. There is a user defined collection of variables that captures the state of the program between subsequent iterations of the loop. The contribution of this paper is a method to construct DSLs with the following properties:

- Our DSL is extendable without requiring changes to existing code. This is achieved by a DSL consisting of a set of type classes. This is enable us to add tailor made objects to control shields. The C++ class associated with such an shield models the use of this shield. In our DSL we add a class of functions that mimics the constructor and methods in C++.
- We use type constructor classes with an argument for the type of the expression in the DSL. In this way expressions in the DSL are typed by the host language, the effect is similar to the types in Generalized Algebraic Data Types, GADTs, [6,14].
- We use functions in the host language to introduce variables in the DSL. The variables themselves are represented by an instance of the type constructor class. This ensures that only defined variables are used. The type system of the host language guarantees the well-typed use of DSL variables.
- We use phantom types to ensure that only a subset of expressions on the left-hand side of an assignment are allowed. This eliminates the need of special `read` and `write` operations for variables.

To the best of our knowledge this combination of properties is not realized by any DSL implementation technique. We show that an architecture based on type classes is able to match all these requirements.

In Sect. 2 we will highlight the aspects of the Arduino that are used in this paper. The design considerations of our DSL are discussed in Sect. 3. Section 4 introduces our actual DSL. In Sect. 5 we show how this DSL is compiled to C++ code. In its turn this is compiled to binary code for the Arduino using the standard Arduino IDE. To show that our DSL can be interpreted in different ways, has multiple *views*, we show in Sect. 6 how a program in our DSL is simulated

[1] DSL = Domain Specific Language, also called DSEL for Domain Specific Embedded Language.

by the iTask system. In Sect. 7 we show how the DSL can be extended with the necessary operations to handle shields. In Sect. 8 we compare our work to relevant related work. We draw conclusions in Sect. 9.

2 The Arduino

The Arduino is a small board with a microprocessor and many input/output ports. The standard programming environment allows us to write Arduino programs in a dialect of C/C++. Since there is no operating system and there are no other threads, our program must provide work for each and every clock cycle of the microprocessor. This is achieved by the function `loop()` that is executed over and over again. The initialization of the system is done by the definition of variables and a function `setup()`.

The *hello world* program for the Arduino blinks the onboard LED at pin 13.

```
boolean ledOn = false;                  // variable definitions
long lastTime = 0;                      // time of last state change

void setup() {
  pinMode(D13, OUTPUT);                 // set pin13 as an output
}
void loop() {
  if (millis() / 500 > lastTime) {      // two steps per second
    ledOn = not ledOn;                  // flip led status
    digitalWrite(D13, ledOn);           // set pin 13 to status
    lastTime += 1;                      // increment the time counter
  }
}
```

We use here the function `millis()` to get the number of milliseconds after starting the microprocessor. A solution using the current time is extendable to several time dependent task for the Arduino and more stable[2] than the more common solution with a `delay(1000)` to stop the entire processing for 1000 ms.

There are many libraries to control special hardware connected to the Arduino. These libraries typically define a class with methods for to control the hardware. An object of this class is used to maintain the state of the hardware and apply the state changes.

The `Servo` library controls the angle of a servo using Pulse Width Modulation, PWM. A physical servo is connected to one of the pins of an Arduino. An `Servo` object is attached to the same pin in the software. The `write` method of this object generates a PWM signal such that the physical servo takes a position with the corresponding angle in degrees. A program that sweeps the servo between 10 and 170 degrees is:

```
#include <Servo.h>                      // include library

Servo s;                                // create an servo object
```

[2] A better program anticipates the overflow of the clock in **long**s after 25 days.

```
int pos = 10;                          // servo position
int step = 1;                          // angle change of servo
long time = 0;                         // time of last state change

void setup() {
  s.attach(A0);                        // attach servo on pin A0 to object
}
void loop() {
  if (millis() / 25 > time) {          // 40 step per second
    time += 1;
    pos += step;
    if (pos > 170 || pos < 10) {       // outside preferred angle range?
      step = -step;                    // turn the direction of the servo
      pos += step;
    }
    s.write(pos);                      // set servo angle in degrees
  }
}
```

These examples describe only a tiny fractions of the possibilities of the Arduino. See for instance the Arduino websites [2,3], for a more complete overview.

3 Design Considerations

Our long term goal is to perform these kind of hardware control from the iTask environment [18]. The iTask environment is a library that supports Task Oriented Programming, TOP. A tasks here is any kind of job to be executed by a human or a machine. The iTask system has primitives to coordinate tasks, to generate web-interfaces for tasks to be executed by humans, and to visualize the state of tasks. The iTask system is implemented as a large library in the functional programming language Clean [20]. It uses generic programming in many places, for instance to generate web-based user interfaces for arbitrary data types.

Running a complete Clean program on an Arduino is unfeasible given the modest processing power of this hardware platform. One option is to run a task in the iTask framework on an ordinary computer that controls the Arduino, for instance via the Firmata protocol [8]. This is the solution implemented by hArduino [12]. We want to execute independent tasks on the Arduino instead of basically execute the task on an ordinary computer that controls the Arduino step by step. These task should only communicate with the iTask system when it is needed for the algorithm of the tasks. This implies that we need a separate program to control the Arduino that cannot be written in full Clean.

Writing the program for the Arduino in another programming language, like C++, than the rest of the TOP program is undesirable. This would imply two source files using different libraries and compilers that need to cooperate smoothly on a single task. Since the iTask system is often used for rapid prototyping, these programs change often. A single source file is very desirable.

The solution introduced here is to define a DSL to control the Arduino. This ARDSL is embedded in Clean, the host language of the iTask system. In contrast

to most DSLs we have to delimit the features our ARDSL inherits from the host language Clean. We need to translate ARDSL to the Arduino and translating too much of Clean to this hardware platform is beyond its capabilities. Hence, we have to be able to distinguish ARDSL parts from the rest of Clean.

Since ARDSL has to interact directly with various kind of hardware and have to compile to compact Arduino code, we have currently chosen a rather low level of abstraction in ARDSL. Others have investigated the possibilities to use functional reactive programming [17], or state machines [21] for this purpose. We have a user defined global state containing typed variables for the entire ARDSL program. This captures the state of the task between various iterations of the `loop`. Inside the ARDSL program we can read and write the pins of the Arduino. Apart from these imperative constructs we have a monadic bind to use the result of one expression in other parts of the program. To prevent issues with the evaluation order, we give ARDSL an eager semantics. Although ARDSL is an embedded DSL with a different character than the host language, we require that the type system of Clean prevents runtime time problems in ARDSL programs.

The main interpretation of ARDSL programs is as code for an Arduino. We will first translate ARDSL program to C++ code. This C++ code is translated by the existing compiler to a binary for the Arduino and loaded to the microprocessor. However, we require a DSL that is suited for multiple interpretations, called *views*, like the simulation in Sect. 6.

In addition the ARDSL must be extendable without breaking existing code. This is used to add the control for specific peripherals by need. Such control is represented by a set of strongly typed functions. Since ARDSL is extendable, we will compose the entire language piece by piece. The Clean type checker guarantees that the required language parts and views needed are available in each and every ARDSL program.

4 Definition of the DSL

Our DSL consists of a set of type constructor classes. All these type constructor classes have two arguments. The first argument is the type of that language construct within ARDSL, or more precisely the representation of that ARDSL type in the host language Clean, e.g. the Clean type `Int` for **int** and **long** in ARDSL or `Bool` for `boolean`. The role of this type argument is very similar to the type argument in a GADT. The second argument of the type class is a phantom type controlling the read/update access of that language element in ARDSL. Each and every element has `Read` access, only parts of the state have `Update` access.

4.1 Standard Values and Operations

First we define constants and standard operations, like addition and equality, in our DSL. The simplest of these type classes is `lit`, for literal. It lifts a value from the Clean domain to ARDSL. The type constraint `type t` ensures that only proper ARDSL types can be lifted from Clean to ARDSL.

```
class lit v where lit :: t → v t Read | type t
```

The type variable v of this class indicates the view of the DSL. A view is here an interpretation of the DSL. We introduce the compilation and simulation views of our DSL below.

The class `arith` contains the arithmetic operators in ARDSL. For many type classes we list only some representative members here. The appendix contains a complete overview of ARDSL. The arithmetic operators require elements of the current view v as argument. These arguments should be of the same type `t`. This type `t` should be a proper type in ARDSL, and we should have the corresponding operation available in Clean for `t`. The read/update access of the arguments is not relevant, they can even be different. The result is a value with `Read` access.

```
class arith v where
    (+.) infixl 6 :: (v t p) (v t q) → v t Read | type, + t
    (-.) infixl 6 :: (v t p) (v t q) → v t Read | type, - t
```

We have added a dot to the name of the operators in ARDSL. Since the type is slightly different from the type of the operator in Clean, `a a->a`, we cannot replace the new operator `+.` by an instance of the native `+` from Clean. In an embedded language both operators have to coexist, hence we introduce new names.

The logical operators take Boolean expressions as arguments and produce a Boolean result.

```
class logical v where
    (|.) infixr 2 :: (v Bool p) (v Bool q) → v Bool Read
    (&.) infixr 3 :: (v Bool p) (v Bool q) → v Bool Read
    ¬.  :: (v Bool p) → v Bool p
```

Equality and ordening of expressions of type `t` in view `v` are expressed by the classes `eq` and `ord` respectively.

```
class eq v where (=.=) infix 4 :: (v t p) (v t q)→v Bool Read | type,Eq t
class ord v where (<.) infix 4 :: (v t p) (v t q)→v Bool Read | type,Ord t
```

These classes show also why it is not possible to make the type of the expressions in ARDSL, `t`, an argument of the type class. We would have a type class like **class** `arith v t` **where** ... In many situations this would be convenient since we can add restrictions on `t` tailored to the view. However, in the class `logical` the type variable `t` would not be used at all. The type system requires that every argument of a type class is used in all of its members. This implies that `t` cannot be a class variable. Hence, any restriction required by one or more views must be specified in definition. This is exactly the same in GADTs.

The conditional expression, `cond c t` **else** `e`, requires a Boolean argument and two other arguments of the same type. We have a shortcut `IF` that allows that these arguments have different types. it produces a `VOID` result. The variant `If` has only a then-part.

```
class ifelse v where
  cond :: (v Bool q) (v t p) Else (v t p) → v t p | type t
```

```
IF   :: (v Bool q) (v t p) Else (v u r) → v VOID Read | type t & type u
If   :: (v Bool q) (v t p) → v VOID Read | type t
```

4.2 Arduino Specific Operations

Our ARDSL has a number of Arduino specific operations. The class `time` provides the `delay` operation and the function `millis` that yields the time in milliseconds.

```
class time v where
    delay  :: (v Int p) → (v Int Read)
    millis ::              (v Int Read)
```

To prevent that one uses illegal pin-numbers we use enumeration types for the digital and analog pins. The enumeration type `PinMode` captures the three possible modes of the `pins`: input, output, and input with an internal pull-up resitor (this ensures that the input of unconnected pins is `High`).

```
:: DigitalPin
   = D0 | D1 | D2 | D3 | D4 | D5 |D6 | D7 | D8 | D9 | D10 | D11 | D12 | D13
:: AnalogPin = A0 | A1 | A2 | A3 | A4 | A5
:: PinMode   = INPUT | OUTPUT | INPUT_PULLUP
```

There is a class `pin` with instances for `DigitalPin` and `AnalogPin`. This can be used to ensure that a specific argument is one of these two kinds of pins.

The class `pinMode` mimics the control of the pin-mode of these pins.

```
class pinmode v where pinmode :: p PinMode → v VOID Read | pin p
```

Note that both arguments are plain values instead of ARDSL expressions that yields a value of these types. It is very well possible to define a version that allows expressions as arguments:

```
class pinmode v where pinmode :: (v p x) (v PinMode y)→v VOID Read | pin p
```

The analog pins have a builtin AD converter. We can read and write 10-bit decimal numbers on these ports. These numbers are represented by ordinary integers values.

```
class analogIO v where
    analogRead  :: AnalogPin → v Int Read
    analogWrite :: AnalogPin (v Int p) → v Int Read
```

Again there is no expression for the port number, just a plain port number is allowed here. However, there is an integer expression for the value to be written.

Digital IO is actually Boolean input and output. We can use any pin for this kind of IO. For the analog pins an appropriate conversion is applied automatically. The type constrains `readPinD` and `writePinD` are needed in the simulation view in Sect. 6.

```
class digitalIO v where
    digitalRead  :: p → v Bool Read | pin, readPinD p
    digitalWrite :: p (v Bool q) → v Bool Read | pin, writePinD  p
```

4.3 Variables and Assignment

Assignments are strongly typed. They require an updatable expression of the same type `t` as the right-hand side as left-hand side.

```
class assign v where
    (=.) infixr 1 :: (v t Update) (v t q) → v t Read | type t
```

The only way to make these updatable expressions in the current version of ARDSL is with the introduction of state variables.

```
class varDef v where
 int     :: ((v Int  Update)→ARDSL (v t p) (v u q))→ARDSL (v t p) (v u q)
 long    :: ((v Int  Update)→ARDSL (v t p) (v u q))→ARDSL (v t p) (v u q)
 boolean :: ((v Bool Update)→ARDSL (v t p) (v u q))→ARDSL (v t p) (v u q)
```

These definitions have a Clean function, usually a λ-expression, as argument. This function puts the variable in the right position. The view `v` determines how the variables look for that interpretation of ARDSL. Note that the variables are defined in an entire ARDSL program, defined in Subsect. 4.4, instead of just an arbitrary piece of ARDSL code.

In addition there is a monadic bind, >==. Although such a bind is compiled to a variable definition, the given type signature determines that these variables cannot be used in assignments. By just changing the `Read` to an `Update`, this design decision is changed. The `:.` operator is the variant of the bind where the right-hand side does not need the result of the left hand side, it is just the semicolon from imperative language.

```
class bind v where
 (>==) infixr 0 :: (v t p)((v t Read)→(v u q))→(v u q) | type t & type u
 (:.)  infixr 0 :: (v t p)                (v u q) →(v u q) | type t & type u
```

The difference of this bind with the vanilla flavored version in a monad is that our operator >== has a `v t Read` as argument of the function on the right-hand side. In the vanilla bind this function has just a plain value `t` as argument. This prevents that arguments 'leak' back from ARDSL to the host language Clean.

There is no need for a return in this class. With a `lit` we can already promote values from the Clean world to the ARDSL world. Each and every expression of type `v t p` can be viewed as an implicit return whenever that is needed.

Other language constructs can be introduced by need. The advantage of an expandable language is that it is always possible to add constructs by need without breaking existing code. The downside of this is that it is also possible to add constructs that spoil the design, there are limited possibilities to prevent these kind of disasters.

The class `ardsl` is just the union of all classes defined above. To write typed program fragments we can just add the class restriction `ardsl v`, instead of listing a restriction for all classes used in this program. We can of course omit the type and let the Clean compiler derive the required type.

4.4 Complete Programs

For a complete program we require that the user writes two of these expressions, one will become the `setup` function and the other one the `loop` function. The record `ArDSL` just combines two named values to a single value.

```
:: ArDSL a b =
    { setup :: a
    , loop  :: b
    }
```

Inside the language implementation we need a few additional fields to record information about the variables defined and libraries imported. We use a new record and the associated conversion function `ardsl :: (ArDSL a b)→ARDSL a b`.

```
:: ARDSL a b =
    { setup1    :: a
    , loop1     :: b
    , defs      :: [Var]
    , includes  :: [String]
    }

:: Var =
    { vname     :: String   // name
    , vnum      :: Int      // number
    , vtype     :: String   // String representation of the type
    , args      :: String   // arguments of definition, used for objects
    , dynv      :: Dyn      // value represented as a generic dynamic
    }
```

The additional fields are filled by variable definitions, instances of the class `varDef` above, and extensions (like the `servo` class) as defined in Sect. 7.

4.5 An Example ARDSL Program

The hello world program from Sect. 2 represented in our DSL reads:

```
helloworld :: ARDSL (a Bool Read) (a VOID Read) | ardsl a
helloworld =
    boolean λledOn.
    int     λlastTime.
    ardsl {
        setup =
          pinmode D13 OUTPUT :.
          digitalWrite D13 (lit False)
        , loop =
          If (millis /. lit 500 >. lastTime) (
            ledOn =. ¬. ledOn >== λb.
            digitalWrite D13 b :.
            lastTime =. lastTime +. lit 1
          )
    }
```

This shows that syntactic overhead for embedding this program in the host language Clean is fortunately very limited. Future versions of ARDSL have a notion of functions and tasks to make programs more concise.

5 Compiling the DSL

The language ARDSL is completely specified by its classes. When we generate code for ARDSL we only have to provide appropriate instances for these classes. It is not necessary to cope with other parts of the host language Clean. When there are Clean expressions within an ARDSL program, they are evaluated at compile time. Clean is just the macro language of ARDSL.

This compiler view translates ARDSL to the Arduino variant of C/C++. Since our DSL adds only a tiny bit of abstraction to the native C++ language of the Arduin, compilation is rather straight forward. First, we introduce the basic building blocks of this view.

5.1 The Compiler View

The classes defined above determine that every view has two arguments: the type of that DSL construct and its read/update type. The compiler view does not use these arguments. It is a function that transforms a compiler state, `CompState`. This compiler state contains the file that is used to collect the generated code. There are two additional items in the compiler state: `idNum` is a counter to generate fresh numbers for identifiers in the compiler, and `indnt` is the current indentation for pretty printed layout in the output file.

```
:: Code a p = Code (CompState → CompState)

:: *CompState =
    { idNum :: !Int
    , indnt :: !Int
    , gcode :: !*File
    }
```

In the compilation view we will have as little direct acces to this data type as possible. We define a small set of manipulation functions that directly access the compiler state. First there is the operator `+.+` to append two changes to the compiler state. It is just function composition.

```
(+.+) infixr 1 :: (Code a p) (Code b q) → Code c r
(+.+) (Code f) (Code g) = Code (g o f)
```

The function `addCode` adds a string to the output file. The function `toCode` gets a value, transforms it to a string wich in its turn is written to the output file.

```
addCode :: String → Code a p
addCode str = Code λs=:{gcode}.{s & gcode = gcode <<< str}

toCode :: a → Code b p | toString a
toCode a = addCode (toString a)
```

The functions `indIndnt` and `decIndnt` increment and decrement the indentation counter in the state. The function `nl` uses this indent to generate the desired amount of indent after a newline in the output file.

```
incIndnt :: Code a p
incIndnt = Code λc=:{indnt}.{c & indnt = inc indnt}
```

The function `genName` generates a fresh identifier name and gives it to the argument of `genName`. This transforms this function to a piece of code.

```
genName :: (String → Code a p) → Code a p
genName f = Code λs=:{idNum}.(unCode (f ("x" + toString idNum))
                                     {s & idNum = idNum + 1})
```

5.2 Instance of ARDSL for the Compilation View

Based on these primitives we define the code generation view. For literals we just have to convert the value to its string representation in C++ syntax and add this string to the output. The function `valCode` takes care of converting a value to code. It is provided by the class `type`.

```
instance lit Code where lit x = addCode (valCode x)
```

Generating code is very similar for all operators. Hence, we introduce some helper functions. The function `codeOp2` generates code for a two argument operator. This requires code for the first argument, the operator itself, and code for the second argument. The prevent binding problems in the generated code we enclose every composed expressions in brackets. Code generating for the arguments is quite simple, they are instances of `Code` and hence compile themselves.

```
codeOp2 :: (Code a p) String (Code b q) → Code c r
codeOp2 x n y = brackets (x +.+ addCode n +.+ y)

brackets :: (Code a p) → Code a p
brackets x = addCode "(" +.+ x +.+ addCode ")"
```

The generation of code for addition and subtraction uses these functions. All other operators are similar and omitted here.

```
instance arith Code where
    (+.) x y = codeOp2 x "+" y
    (-.) x y = codeOp2 x "-" y
```

The code generation for other constructs in ARDSL is similar. We only show the code generation for the class `time` as example.

```
instance time Code where
    delay  x = addCode "delay" +.+ brackets x
    millis   = addCode "millis ()"
```

The code generating for conditionals is slightly more interesting. For `cond` that produces a typed result we generate an inline conditional in C++. For the `IF` that produces a `VOID` we generate an ordinary conditional statement.

```
instance ifelse Code where
    cond c t Else e =
        brackets (c +.+ addCode "?" +.+ incIndnt +.+ nl +.+
                  t +.+ addCode ":" +.+ nl +.+ e +.+ decIndnt)
    IF c t Else e =
        addCode "if" +.+ brackets c +.+
        braces t +.+  addCode "else" +.+ braces e
```

Variables and Assignment. In this target language the handling of variables is very easy. We just have to generate the name of the variable and the Arduino compiler will give it the desired read or update interpretation. When other parts of the compilation ensure that the code for a variable is the adding of its name to the code, the code for assignment becomes just

```
instance assign Code where (=.) v e = v +.+ addCode "=" +.+ e
```

This determines how to generate code for the introduction of variables. The argument given to the function `f` that generates the `ardsl` is just an instance of the code generation that writes the name of the variable to the file: `addCode name`. We do not yet generate code for the variable definition itself, we delay this until we have collected all variable definitions and libraries we have to collect.

```
instance varDef Code where
 int f = {ardsl & defs = [var: ardsl.defs]} where
  ardsl = f (addCode name)
  name  = intToName vnum
  vnum  = newName ardsl
  var   = {vname = name,vnum = vnum,vtype = "int",args = "",dynv = toDyn 0}
```

For a bind operator we generate a variable that is used to store the result of the first argument of the bind. For the right-hand side of the bind we just have to provide code that generates the variable name as argument. The name of the new variable is generated by `genName` discussed above.

```
instance bind Code where
  (>=) x f =
    genName λv.addCode (typeCode (code2val x) + "" + v + "=") +.+ x +.+
               addCode ";" +.+ nl +.+ f (addCode v)
```

The function `compile :: *File (ARDSL (Code a p) (Code b q)) → *File` completes the compilation view of ARDSL. This function writes a header to the given file, define the imports and variables from the give program followed by the code for the given `setup` and `loop`.

Example. The code generation described here applied to the example in Subsect. 4.5 produces:

```
boolean v1;
int v0;
```

```
void setup() {
    pinMode (13, OUTPUT);
    digitalWrite (13, false);
}
void loop() {
    if (((millis () / 500) > v0)) {
        boolean x2 = v1 = (not v1);
        digitalWrite (13, x2);
        v0 = (v0 + 1);
    };
}
```

6 Simulating the DSL

To demonstrate that our DSL can have multiple interpretations we develop a simulator for ARDSL programs. The simulator is able to apply the `setup` or the `loop` function to the state of the simulator. In order to do this we need a tailor made state containing the variables of the ARDSL program and view that translates the ARDSL program to a state manipulating function.

6.1 Simulation State

The simulation state contains the values of all variables in the global state of the program, the value of all input/output pins used and the current time.

```
:: State =
  { vars  :: [(String, Dyn)]
  , dpins :: [(DigitalPin, Bool)]
  , apins :: [(AnalogPin, Int)]
  , time  :: Int
  }
```

Variable are indicated by their index in the list `vars`. The string in the tuple is the name used in the ARDSL program, it is included just as a reference. Since there are variables of different types, we store their value as a dynamic of type `Dyn`. In the type `Dyn` every value is represented as a list of strings. There are generic [1] functions for the transformation to and from `Dyn`. We use this tailor-made type instead of the type `Dynamic` from Clean to ensure the values are human readable during the simulation.

The variables in our DSL need some special treatment in this translation. The interpretation of variables depends on the context where they are used. In all usual cases a variable should be replaced by its actual value in a simulator. In the left-hand side of an assignment however, a variable indicated the position is the state that needs to be updated. This is handled by the type `RW` that is given as additional argument to the state transition function. The `R` indicates a read action, `W a` is a write of the given value, and `F f` applies the given function to the value in the state[3].

[3] Every `W a` can be replaced by `F λ_.a`, but `W a` is a convenient shortcut.

```
:: RW a = R | W a | F (a → a)
```

The function `readVar` reads a variable from the state and applies the indicated action. Just a read for `W`, a replacement by the given value for `W a`, or an application of the given function followed by an update for `F f`.

```
readVar :: VarId (RW a) State → (a, State) | dyn a
readVar v R s=:{vars} = (fromJust (fromDyn (snd (vars !! v))), s)
readVar v (W a) s=:{vars}
    = (a, {s & vars = updateAt v ((fst (vars !! v)), toDyn a) vars})
readVar v (F f) s=:{vars}
    # (n,d) = vars !! v
    # o = f (fromJust (fromDyn d))
    = (o,{s & vars = updateAt v (n, toDyn o) vars})
```

For the simulator we make an instance of the type `Eval` for all classes in ARDSL. This is just a state transformer that takes the read/write type `RW` as additional argument.

```
:: Eval a p = Eval ((RW a) State → (a, State))
```

For convenience we define an instance of the class `monad` for this type.

```
instance monad Eval where
    (>>=) (Eval x) f = Eval λrw s.let (a, t) = x R s in unEval (f a) R t
    rtrn a = Eval λ_ s.(a, s)
```

6.2 Simulation View

The instances for this view for all standard functions are the usual monadic definitions of an evaluator. For a literal constant we just return the given value.

```
instance lit Eval where lit x = rtrn x
```

All operators are eager. They evaluate their arguments and return the computed value. For instance the arithmetic operators are defined as:

```
instance arith Eval where
    (+.) x y = eval2 x y (+)
    (-.) x y = eval2 x y (-)

eval2 :: (m a p) (m b q) (a b→c) → m c r | monad m
eval2 x y f = x >>= λa. y >>= λb. rtrn (f a b)
```

For the conditional expressions we evaluate the boolean expression. Based on its value the then or else part is evaluated. For the `If` and `IF` we apply the desired type conversions.

```
instance ifelse Eval where
cond c t Else e = c >>= λb. if b t e
IF c t Else e = c >>= λb. if b (t >>=λa.rtrn VOID)(e >>= λa.rtrn VOID)
If c t        = c >>= λb. if b (t >>= λa.rtrn VOID) (rtrn VOID)}
```

The instance for variable definitions is surprisingly similar to the definition for code generation shown above. We update the administration inside the ARDSL to record the definition of a new variable. The difference between the previous instance and this one is that we replace each applied occurrence of the variable by a `readVar vnum` in the simulator.

```
instance varDef Eval where
 int f = {ardsl & defs = [var: ardsl.defs]} where
  ardsl = f (Eval (readVar vnum))
  name  = intToName vnum
  vnum  = newName ardsl
  var   = {vname = name,vnum = vnum,vtype = "int",args = "",dynv = toDyn 0}
```

Other operations manipulating the state, i.e. the read and write of pins and accessing the time, are implemented with a similar access to the state.

After these preparations the implementation of the assignment is simple. We just have to give the new value to the `readVar vnum` function inside the variable. The new value is obtained by evaluating the right-hand side.

```
instance assign Eval where (=.) v e = e >>= λa.Eval λr s.unEval v (W a) s
```

For this view of the bind operator we do not need to introduce a variable in the state. Since these variable introduced here is read-only, we can use a λ-expression to directly support its value to all applied occurrences. This is exactly what the monad of this simulation view does:

```
instance bind Eval where (>==) x f = x >>= f o rtrn
```

This completes the simulation view of our ARDSL.

6.3 iTask Simulator

Using the simulation view of ARDSL programs is very easy to make a simple simulator for programs in the iTask system. The state is just a data structure. The generic mechanism of the iTask system is able to generate a web-based display and editor for this data structure by `derive class iTask State`. In the task it is initially possible to apply the `setup` function. In the next state we can either apply the loop, or a reset function. The editor allows the user to inspect and change the state between steps. There is a shutdown button in each state.

```
simulate :: (ARDSL (Eval a p) (Eval b q)) → Task ()
simulate {setup1=(Eval codesetup),loop1=(Eval codeloop),defs} = simsetup
where
 simsetup =
  updateInformation "State" [] s0 >>*
   [ OnAction ActionFinish (always shutDown)
   , OnAction (Action "setup" []) (hasValue (simloop o snd o codesetup R))
   ]
 simloop s =
  updateInformation "State" [] s >>*
```

```
[ OnAction ActionFinish (always shutDown)
, OnAction (Action "Reset" []) (always simsetup)
, OnAction (Action "loop" []) (hasValue (simloop o snd o codeloop R))
]
s0 = { vars  = [(v.vname, v.dynv) \\ v ← reverse defs]
     , dpins = [] , apins = [] , time  = 0}
```

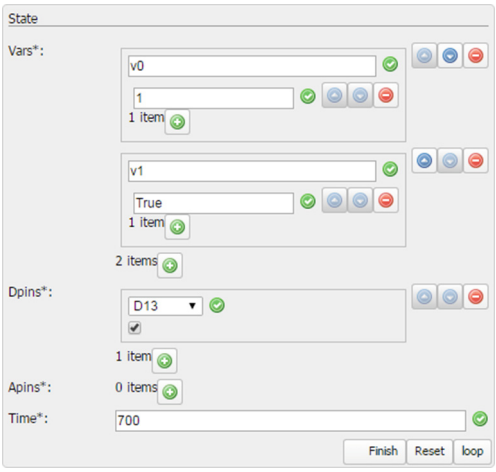

Fig. 1. Screenshot of the simulation of the blinking LED example.

Figure 1 shows a screenshot of the simulation of our hello world example. The translation generated the name `v0` for variable `time` and `v1` for the Boolean `ledOn` respectively. Output `D13` is `True` or *high* to indicate that the LED is on.

7 Extending the DSL

The entire design of ARDSL is modular and prepared to be extended. Adding a additional class is nothing new for our DSL. We use this mechanism to add a typed representation of C++ class controlling a shield to our DSL.

The servo library is a simple example. It contains a function `servo` to define a `Servo` object. It is very similar to the definition of variables in the state. The most notable difference is that the read/update attribute is `Read`, this ensures that it is not possible to update a servo variable. Each servo object has two methods, or manipulation functions. These functions have (a reference to) the servo object as their first argument. The `attach` function connects the servo to the specified pin. The `writeS` function sets the servo to the given position in degrees.

```
:: Servo = {pin :: String, pos :: Int}

class servo x where
```

```
attach :: (x Servo q) p → x VOID Read | pin p
writeS :: (x Servo q) (x Int q) → x VOID Read
servo  :: ((x Servo Read)→ARDSL (x a p) (x b q)) → ARDSL (x a p) (x b q)
```

An ARDSL program that sweeps the servo gently between 10 and 170 degrees is:

```
servosweep :: ARDSL (a VOID Read) (a VOID Read) | ardsl, servo a
servosweep =
    servo λs.
    int   λpos.
    int   λstep.
    long  λtime.
    ardsl {
    setup =
        pos =. lit 10 :.
        step =. lit 1 :.
        attach s A5
    , loop =
        If (millis /. lit 25 >. time) (
            time =. time +. lit 1 :.
            pos  =. pos +. step :.
            If (pos >. lit 170 |. pos <. lit 10) (
                step =. lit 0 -. step
            ) :.
            writeS s pos
        )
    }
```

The record `Servo` contains the fields necessary to simulate a servo, it is the value of a servo in ARDSL. This servo object becomes one of the variables in the `ARDSL` record. Implementing the views `Comp` and `Eval` for this class follows exactly the pattern of the other classes. It contains no surprises. by deriving an iTask representation for it, we can include a servo directly in the simulation. It is possible to make a custom view of a servo in the simulation.

In the same style we can add other libraries, for instance to drive liquid crystal displays. LCDs are often used to improve the output potions of an Arduino.

8 Related Work

There is very much work done in the field of DSLs embedded in functional programming languages. Back in 1998 Paul Hudak described a way to make modular DSLs with a single view [13]. The next year Leijen and Meijer described a way to compile a DSL [16]. Elliott wrote one of the many improvements of these techniques [7]. Augustsson actively contributes to the development of implementations of a DSL embedded in a functional programming language, e.g. [4]. A more recent step in the development is described by Gill [11]. In his CEFP paper Gibbons delivers a recent overview of deep and shallow embedding techniques to build a DSL [9]. He also contribute to the efficient implementation of DSLs by folding [10].

The particular multi-view strongly typed extendable way to make a DSL described in this paper is to the best of our knowledge new. Dynamically typed variants implementing a version of the λ-calculus are available [5,15].

9 Conclusion

This paper introduces new techniques to make an embedded DSL. By using a set of classes as DSL, it is extendable. We can always add a class. By making a new instance of these classes we make a new view of our DSL. By using a type constructor class with the type of the DSL component as argument, the type system of the host language checks the types of the DSL. An additional phantom type argument of the type constructor class controls the desired read write behavior of DSL fragments. The approach to construct a DSL only requires type constructor classes in the host language. A DSL that is based on data types, deep embedding, requires GADTs for type safety inside the DSL. These GADTs must be extendable to handle the addition of new libraries without breaking existing code. This are much higher demands on the host language.

We use function in the host language to let the system introduce variables in the DSL. In this way we can assure that all variables used in DSL program exist and are properly typed. This paper illustrates these techniques in a DSL to program Arduino microprocessors hosted in Clean.

Since the host language compiler does most of the work, the compile and simulate view are very efficient. They only have to execute the functions selected by the Clean compiler. Due to the low abstraction level of ARDSL, the generated C++ code is as compact and efficient as the corresponding C++ code. Future improvements of this DSL should enhance the abstraction level to make programs more task oriented and compact. This will be achieved by the introduction of a notion of recursive functions and tasks.

Acknowledgement. We thank the reviewers for their valuable feedback.

Appendix

The classes defining the core of ARDSL.

```
class lit v where
    lit     :: t → v t Read | type t
class arith v where
    (+.) infixl 6 :: (v t p) (v t q) → v t Read | type, + t
    (-.) infixl 6 :: (v t p) (v t q) → v t Read | type, - t
    (*.) infixl 7 :: (v t p) (v t q) → v t Read | type, * t
    (/.) infixl 7 :: (v t p) (v t q) → v t Read | type, / t
    (%.) infixl 7 :: (v t p) (v t q) → v t Read | type, rem t
class logical v where
    (|.) infixr 2 :: (v Bool p) (v Bool q) → v Bool Read
    (&.) infixr 3 :: (v Bool p) (v Bool q) → v Bool Read
```

```
    ¬.  :: (v Bool p) → v Bool p
class eq v where
    (=.=) infix 4 :: (v t p) (v t q) → v Bool Read | type, Eq t
    (!.=) infix 4 :: (v t p) (v t q) → v Bool Read | type, Eq t
class ord v where
    (<.)  infix 4 :: (v t p) (v t q) → v Bool Read | type, Ord t
    (<.=) infix 4 :: (v t p) (v t q) → v Bool Read | type, Ord t
    (>.)  infix 4 :: (v t p) (v t q) → v Bool Read | type, Ord t
    (>.=) infix 4 :: (v t p) (v t q) → v Bool Read | type, Ord t
class bind v where
 (>==) infixr 0 :: (v t p)((v t Read)→(v u q))→(v u q) | type t & type u
 (:.)  infixr 0 :: (v t p)              (v u q) →(v u q) | type t & type u
class digitalIO v where
    digitalRead  :: p → v Bool Read | pin, readPinD p
    digitalWrite :: p (v Bool q) → v Bool Read | pin, writePinD  p
class analogIO v where
    analogRead  :: AnalogPin → v Int Read
    analogWrite :: AnalogPin (v Int p) → v Int Read
class time v where
    delay  :: (v Int p) → (v Int Read)
    millis ::              (v Int Read)
class pinMode v where
    pinmode :: p PinMode → v VOID Read | pin p
class assign v where
    (=.) infixr 1 :: (v t Update) (v t q) → v t Read | type t
class ifelse v where
    cond:: (v Bool q) (v t p) Else (v t p) → v t p | type t
    If  :: (v Bool q) (v t p) → v VOID Read | type t
    IF  :: (v Bool q) (v t p) Else (v u r) → v VOID Read | type t & type u
class varDef v where
 int     :: ((v Int Update) →ARDSL (v t p) (v u q))→ARDSL (v t p) (v u q)
 long    :: ((v Int Update) →ARDSL (v t p) (v u q))→ARDSL (v t p) (v u q)
 boolean :: ((v Bool Update)→ARDSL (v t p) (v u q))→ARDSL (v t p) (v u q)

class ardsl v | lit, arith, logical, eq, ord, bind, digitalIO, analogIO,
                time, pinMode, assign, ifelse, varDef v

class type t where
    type     :: t → String
    typeCode :: t → String
    valCode  :: t → String
instance type Int, Bool, Real, Char, VOID, String, DigitalPin, AnalogPin
```

References

1. Alimarine, A.: Generic Functional Programming: Conceptual Design, Implementation and Applications. UB Radboud University Nijmegen (2003)
2. arduino.cc website (2015). www.arduino.cc

3. arduino.org website (2015). www.arduino.org
4. Augustsson, L.: Making edsls fly (2012). http://vimeo.com/73223479
5. Carette, J., Kiselyov, O., Shan, C.C.: Finally tagless, partially evaluated: tagless staged interpreters for simpler typed languages. J. Funct. Program. **19**(5), 509–543 (2009)
6. Cheney, J., Hinze, R.: First-class phantom types. Technical report, Cornell University (2003)
7. Elliott, C., Finne, S., de Moor, O.: Compiling embedded languages. J. Funct. Program. **13**(2), 455 (2003). Updated version of paper by the same name that appeared in SAIG2000 Proceedings. http://conal.net/papers/jfp-saig/
8. Firmata protocol (2015). http://firmata.org/wiki/Main_Page, http://github.com/firmata/protocol
9. Gibbons, J.: Functional programming for domain-specific languages. In: Zsók, V., Horváth, Z., Csató, L. (eds.) CEFP 2013. LNCS, vol. 8606, pp. 1–28. Springer, Heidelberg (2015)
10. Gibbons, J., Wu, N.: Folding domain-specific languages: deep and shallow embeddings (functional pearl). SIGPLAN Not. **49**(9), 339–347 (2014)
11. Gill, A.: Domain-specific languages and code synthesis using haskell. Queue **12**(4), 30:30–30:43 (2014)
12. hArduino.: package that allows haskell programs to control arduino boards using the Firmata protocol (2013). http://hackage.haskell.org/package/hArduino
13. Hudak, P.: Modular domain specific languages and tools. In: Proceedings of the 5th International Conference on Software Reuse. ICSR 1998, p. 134. IEEE Computer Society, Washington DC (1998)
14. Johann, P., Ghani, N.: Foundations for structured programming with gadts. SIGPLAN Not. **43**(1), 297–308 (2008)
15. Koopman, P.: Functional semantics. In: Achten, P., Koopman, P. (eds.) The Beauty of Functional Code. LNCS, vol. 8106, pp. 60–78. Springer, Heidelberg (2013)
16. Leijen, D., Meijer, E.: Domain specific embedded compilers. SIGPLAN Not. **35**(1), 109–122 (1999)
17. Lindberg, R.: fpr-arduino: Functional Reactive Programming for the Arduino (2015). http://github.com/frp-arduino
18. Plasmeijer, R., Achten, P., Koopman, P.: iTasks: executable specifications of interactive work flow systems for the web. In: Hinze, R., Ramsey, N. (eds.) Proceedings of the ICFP 2007, pp. 141–152. ACM, Freiburg (2007)
19. Plasmeijer, R., Lijnse, B., Michels, S., Achten, P., Koopman, P.: Task-oriented programming in a pure functional language. In: Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming. PPDP 2012, pp. 195–206. ACM, New York (2012)
20. Plasmeijer, R., van Eekelen, M.: Clean language report (version 2.1) (2002). http://clean.cs.ru.nl
21. Schwaab, C., Pfeiffer, M., Brady, E.: Safety first: targeting embedded systems with full-spectrum dependent types, TFP 2015 Draft Proceedings (2015)

Programmable Signatures

Anders Persson(✉) and Emil Axelsson

Chalmers University of Technology, Gothenburg, Sweden
anders.persson@chalmers.se

Abstract. When compiling Embedded Domain Specific Languages (EDSLs) into other languages, the compiler translates types in the source language into corresponding types in the target language. The translation is often driven by a small set of rules that map a single type in the source language into a single type in the target language. This simple approach is limiting when there are multiple possible mappings, and it may lead to poor interoperability and poor performance in the generated code. Instead of hard-wiring a single set of translation rules into a compiler, this paper introduces a small language that lets the programmer describe the mapping of each argument and function separately.

1 Introduction

Feldspar is an embedded domain specific language written in Haskell [1,2].[1] The purpose of Feldspar is to implement high-performance software, especially in the domain of signal processing in embedded systems [3].

Feldspar comes with an optimizing compiler that translates Feldspar expressions into C99 code.[2] When translating a function signature, the compiler uses a specific calling convention as detailed in Chap. 1.4.2 in reference [3]:

- All functions return `void`
- Scalar values are passed by value
- Structured values (structs, arrays) are passed by reference
- Arrays are represented using a data structure `struct array`
- Return values are passed through caller provided pointers

For example, the following is the type of an FFT function in Feldspar:

```
1 fft :: Data [Double] -> Data [Double]
```

The type constructor `Data` denotes a Feldspar expression, and its parameter denotes the type of value computed by that expression. For historical reasons, Feldspar uses `[]` to denote immutable arrays.

The compiler translates the `fft` function into the following C99 signature,

```
1 void fft(struct array *v0, struct array **out);
```

[1] https://hackage.haskell.org/package/feldspar-language.
[2] https://hackage.haskell.org/package/feldspar-compiler.

M. Serrano and J. Hage (Eds.): TFP 2015, LNCS 9547, pp. 124–135, 2016.
DOI: 10.1007/978-3-319-39110-6_7

where **struct** array is a Feldspar specific data structure with metadata, such as the number of elements, and a pointer to the data area.

The Feldspar compiler uses its calling convention for a number of reasons, but the primary reasons are consistency and generality. The convention ensures that all arguments fit into a register, which helps avoid spilling arguments to the call stack. By passing arrays as references bundled with their length, the compiler can generate code that works with different array sizes and still preserve the same number of arguments.

Note that the calling convention only applies to the main function that one wants to compile. Due to the embedding in Haskell, any helper functions that are used will be inlined into the main function before code generation. Take the following Feldspar program as an example:

```
1  f :: Data Double -> Data Double
2  f x = x*2
3
4  g :: Data Double -> Data Double
5  g y = f (y+1)
```

Compiling the function `g` is equivalent to compiling the function `\y -> (y+1)*2`.

In the future, we want to allow the possibility of marking certain Feldspar functions as non-inlineable. But in that case, we expect that the compiler will automatically handle the internal calling conventions.

1.1 Issues with Fixed Mappings

With a fixed signature mapping, it is easy to derive the target language type from the source language type. But the fixed mapping leaves little room to change the generated signature to fit into existing software. Code generated from Feldspar will usually be part of a larger system, and the calling convention is naturally dictated by the system rather than by the Feldspar compiler. If the calling convention of the system differs from that of Feldspar, glue code has to be written to interface with functions generated from Feldspar.

In a typical embedded system, arrays are passed as two arguments: a pointer to the data buffer and an integer that gives the number of elements of the array. However, there are many variations on this theme. Should the length come before or after the buffer? Can the length argument be used for more than one array if they always have the same length? And so on.

Even if we allow flags to customize the compiler, a fixed set of mapping rules will never be able to cover all possible situations. Instead, we would like to put the exported signature in the hands of the programmer. As a concrete example, take the following function for computing the scalar product of two vectors:

```
1  scProd :: Data [Double] -> Data [Double] -> Data Double
```

The generated signature with the default mapping is:

```
1 void scProd(struct array *v0, struct array *v1, double *out);
```

By default, the Feldspar compiler automatically makes up names for the arguments. Apart from the problem that Feldspar's `struct array` is an unconventional array representation, this code may also be considered too general: it has to cater for the fact that the arrays may have different lengths. Since it does not make sense to call `scProd` with arrays of different lengths, a more appropriate signature might be:

```
1 double scProd(uint32_t len, double *v1_buf, double *v2_buf);
```

Here, the arrays are passed as two pointers to the corresponding data buffers and a single length argument. This signature is more likely to occur in a practical system, and it has the advantage that the function does not have to decide what to do if the lengths are different. However, the system may expect a different order of the arguments, and might expect the result to be passed by value instead of by reference.

In addition to being able to customize the calling convention, we might also want to affect non-functional aspects of functions. For example, we can name arguments for readability and debugging purposes. This is helpful since Feldspar is an embedded language and that syntactic information is lost when the Haskell compiler reads the source file.

In future work we want to extend the annotations to include attributes to help the C compiler, including `restrict` and `volatile`.

1.2 Contributions

To address the problems above, this paper presents three contributions:

- We define a simple EDSL to specify type conversions and annotations when exporting a Feldspar function to an external system (Sect. 2).
- We give an implementation of the EDSL as a small wrapper around the existing Feldspar compiler (Sect. 3). The implementation relies on a simple interface to the underlying compiler.
- A generalized version of the implementation and the interface are provided as part of the `imperative-edsl`[3] package.

2 The Signature Language

Dissatisfied with hard-wired rules and global compiler options, we propose a small language as a more flexible way to drive the compiler.

[3] https://hackage.haskell.org/package/imperative-edsl-0.4.

```
-- | Capture an argument
lam :: (Type a) => (Data a → Signature b) → Signature (a → b)

-- | Capture and name an argument
name :: (Type a) => String → (Data a → Signature b) → Signature (a → b)

-- | Create a named function return either by value or reference
ret :: (Type a) => String → Data a → Signature a
ptr :: (Type a) => String → Data a → Signature a
```

Listing 1.1. Signature language

The Signature language allows the programmer to express the mapping of individual arguments separately. Specifically it allows the programmer to add annotations to every argument and control the data representation. These annotations can be as simple as just giving a name to a parameter, using the `name` combinator. Or, it can change the arity of the function by introducing new parameters, like the `native` and `exposeLength` combinators in Listing 1.2.

Like Feldspar, the Signature language is a typed domain specific language, embedded in Haskell. The Signature language preserves the type safety of Felspar.

The Signature language interface is given in Listing 1.1. The combinators `lam` and `name` are used to bind (and possibly annotate) an argument, while `ret` and `ptr` are used to return the result of the function to be generated.

As our running example, we will reuse the `scProd` function from Sect. 1.1.

```
scProd :: Data [Double] -> Data [Double] -> Data Double
```

We can mimic the standard rules of the Feldspar compiler by wrapping the function in our combinators.

```
ex1 :: Signature ([Double] -> [Double] -> Double)
ex1 = lam $ \xs -> lam $ \ys -> ptr "scProd" (scProd xs ys)
```

which generates the following C signature when compiled

```
void scProd(struct array *v0, struct array *v1, double *out);
```

Using `name` instead of `lam`, we change the embedding to name the first argument

```
ex2 :: Signature ([Double] -> [Double] -> Double)
ex2 = name "xs" $ \xs -> lam $ \ys  -> ptr "scProd" (scProd xs ys)
```

resulting in

```
void scProd(struct array *xs, struct array *v1, double *out);
```

Finally, we change the function to return by value, by using `ret` instead of `ptr`

```
ex3 :: Signature ([Double] -> [Double] -> Double)
ex3 = name "xs" $ \xs -> name "ys" $ \ys -> ret "scProd" (scProd xs ys)
```

which produces

```
1  double scProd(struct array *xs, struct array *ys);
```

The basic constructors in the language are useful for simple annotations on the arguments. But it is also possible to create constructors that will change the arity or introduce interface code into the embedded function. The interface code can bridge different representation formats.

Without the `Signature` language, we would have to write a C wrapper around the generated function. A wrapper written in C is not polymorphic, but declared with concrete types, like **`int`** or **`double`**. In contrast, the Feldspar functions are often polymorphic and the concrete types are decided at compile time. A hand-written wrapper would have to change for different concrete types, and thus becomes a maintenance burden. Also, the wrapper code is a separate function and can not be optimized together with the generated code. In contrast, the `Signature` language combinators are applied before optimization and code generation, and the wrapper code fuses with the function.

For example, consider the `scProd` function again. In earlier versions it suffered from two problems.

1. The two arrays may have different lengths and the generated code has to defensively calculate the minimum length (see line 6 below).
2. The arrays are passed using a **`struct`** `array` pointer which results in extra dereferencing. On line 9 below `at` is a macro that indexes into a **`struct`** `array` and to do that it must do an extra dereference to find the buffer.

```
1  void scProd(struct array *v0, struct array *v1, double *out)
2  {
3      double e4;
4      uint32_t len5;
5
6      len5 = min(getLength(v0), getLength(v1));
7      e4 = 0.0;
8      for (uint32_t v2 = 0; v2 < len5; v2 += 1) {
9          e4 = e4 + at(double, v0, v2) * at(double, v1, v2);
10     }
11     *out = e4;
12 }
```

To help alleviate these problems we can define smart constructors that modify the code before optimization. Note that these smart constructors are extensions to the `Signature` language and can be expressed by the end user.

The `native` function changes an array argument to a native C array with length `l`. `Lam` is a constructor of the `Signature` type (see Sect. 3). It is like `lam`, except that it takes an extra annotation as argument. In this case, the annotation `Native l` says that the argument bound by `Lam` should be a native C array of length `l`. By using the Feldspar `setLength` function, size information is added to the array arguments, so that the function `f` can use the argument as an ordinary Feldspar array that has an associated length.

```
-- | Pass the argument as a native array of length @len@
native :: (Type a)
       => Data Length → (Data [a] → Signature b) → Signature ([a] → b)
native l f = Lam (Native l) $ λa → f $ setLength l a

-- | Expose the length of an array
exposeLength :: (Type a)
             => (Data [a] → Signature b) → Signature (Length → [a] → b)
exposeLength f = name "len" $ λl → native l f
```

Listing 1.2. Smart signature constructors

In Sect. 3 we show how the `Native` constructor produces the interface code needed to translate between native and **`struct`** `array` formats.

The `exposeLength` function adds an extra length argument to the signature and passes this length to `native`. The effect is to break up a standard array argument into two arguments: a length and a native array.

With our new combinators, we can create a version of the `scProd` function that accepts native arrays of a fixed (runtime specified) length

```
scProdNative = name "len" $ \len ->
               native len $ \as  ->
               native len $ \bs  ->
               ret "scProd" $ scProd as bs
```

which compiles to:

```
double scProd(uint32_t len, double *v1_buf, double *v2_buf)
{
    struct array v2 = {.buffer =v2_buf, .length =len, .elemSize =
                       sizeof(double), .bytes =sizeof(double) * len};
    struct array v1 = {.buffer =v1_buf, .length =len, .elemSize =
                       sizeof(double), .bytes =sizeof(double) * len};
    double e5;

    e5 = 0.0;
    for (uint32_t v3 = 0; v3 < len; v3 += 1) {
        e5 = e5 + at(double, &v1, v3) * at(double, &v2, v3);
    }
    return e5;
}
```

Note how the Feldspar compiler now realizes that both vectors have the same length, and thus removes the defensive minimum length calculation.

The first two declarations in the generated code are for converting the native array in the interface to **`struct`** `array` which is what the body of the function expects. When the **`struct`** `arrays` are allocated on the stack and not visible outside the function, an optimizing C compiler can often remove the extra dereference. Instead of relying on compiler optimizations, we plan to make it possible to use native arrays throughout the generated code, when stated so in the signature, but that requires a change to the Feldspar compiler and is out of scope for this paper.

3 Implementation

The language is implemented as a simple deep embedding (Listing 1.3) on top of which the programmer interface in Listing 1.1 is defined. The simplicity of the deep embedding means that the compiler has a small set of constructs to deal with. Still it supports the definition of a richer interface to the user. For example, the `exposeLength` function could be implemented entirely in terms of simpler constructs. This way of combining a deep embedding with shallow user-facing functions has been shown to be very powerful for implementing EDSLs [4].

We can think of `Signature` as adding top-level lambda abstraction and result annotations to the existing expression language `Data`. The use of a host-language function in the `Lam` constructor is commonly known as *higher-order abstract syntax* (HOAS) [5]. HOAS allows us to construct signatures without the need to generate fresh variable names. As we will see in section Sect. 3.1, names are instead generated when we generate code from the signature.

In this paper we show the `Signature` implementation specialized to the Feldspar language. A generalized version of the implementation is provided as part of the `imperative-edsl` library.

3.1 Code Generation

`Signature` is defined as a wrapper type around the Feldspar expression type `Data`. In order to generate code for signatures, we first need to be able to generate code for `Data`. To this end, the Feldspar compiler provides the following interface:

```
varExp    :: Type a             => VarId -> Data a
compExp   :: (MonadC m)         => Data a -> m C.Exp
compTypeF :: (MonadC m, Type a) => proxy a -> m C.Type
```

The first function, `varExp`, is used to create a free variable in Feldspar. Naturally, this function is not exported to ordinary users. The function `compExp` is used to compile a Feldspar expression to a C expression `Exp`. Since compilation normally results in a number of C statements in addition to the expression, `compExp` returns in a monad `m` capable of collecting C statements that can later be pretty

```
-- | Annotations to place on arguments or result
data Ann a where
  Empty  :: Ann a
  Native :: Type a => Data Length → Ann [a]
  Named  :: String → Ann a

-- | Annotation carrying signature description
data Signature a where
  Ret    :: (Type a) => String → Data a → Signature a
  Ptr    :: (Type a) => String → Data a → Signature a
  Lam    :: (Type a)
         => Ann a → (Data a → Signature b) → Signature (a → b)
```

Listing 1.3. Signature Language (deep embedding)

printed as C code. Finally, `compTypeF` is used to generate a C type from a type `a` constrained by Feldspar's `Type` class. The argument of type `proxy a` is just used to determine the type `a`.

The code generator is defined in Listing 1.4. Before explaining how it works, we will explain the code generation technique used.

We use a C code generation monad for producing the C code. Operations of this monad are accessed via the `MonadC` type class. Among other things, it provides a method for generating fresh names, methods for adding statements to the generated code and for adding parameters to the currently generated function definition.

The concrete pieces of C code to be generated are written as actual C code using quasi-quoters [6] for C code, provided by the package `language-c-quote`[4].

For example, consider the following two lines from Listing 1.4:

```
17          addParam [cparam| $ty:t *out |]
18          addStm [cstm| *out = $e; |]
```

The first line adds a parameter to the generated C function, and the second line adds a statement that assigns the result to the output pointer. The `[q| ... |]` syntax is for quasi-quotation, where `q` is the name of the quoter. The quoter parses the C code inside the brackets, and turns it into a representation of a piece of code that can be collected in the code generation monad.

Quasi-quoters also allow the splicing of Haskell values into the quoted code. In the above example, `$ty:t` splices in the Haskell value `t` as a C type, and `$e` splices in `e` as a C expression. For the code to type check, `t` must have the type `C.Type` and `e` must have the type `C.Exp`.

The signature is compiled by recursively traversing the `Lam` constructors and building up the argument list. Finally, the `Ret` or `Ptr` case combines the arguments to produce the function signature. The compilation of the function body is delegated to the Feldspar compiler (by calling `compExp`).

The `Lam (Native l)` case (lines 24–38 from Listing 1.4) is an example of how the `Signature` language can generate interface code.

```
24      go fun@(Lam n@(Native l) f) prelude = do
25        t <- compTypeF (elemProxy n fun)
26        i <- freshId
27        let w = varExp i
28        C.Var (C.Id m _) _ <- compExp w
29        let n = m ++ "_buf"
30        withAlias i ('&':m) $ go (f w) $ do
31          prelude
32          len <- compExp l
33          addLocal [cdecl| struct array $id:m = { .buffer = $id:n
34                                                , .length=$len
35                                                , .elemSize=sizeof($ty:t)
36                                                , .bytes=sizeof($ty:t)*$len
37                                                }; |]
38          addParam [cparam| $ty:t * $id:n |]
```

[4] http://hackage.haskell.org/package/language-c-quote.

```
-- | Compile a @Signature@ to C code
translateFunction :: forall m a. (MonadC m) => Signature a → m ()
translateFunction sig = go sig (return ())
  where
    go :: forall d. Signature d → m () → m ()
    go (Ret n a) prelude = do
      t ← compTypeF a
      inFunctionTy t n $ do
        prelude
        e ← compExp a
        addStm [cstm| return $e; |]
    go (Ptr n a) prelude = do
      t ← compTypeF a
      inFunction n $ do
        prelude
        e ← compExp a
        addParam [cparam| $ty:t *out |]
        addStm [cstm| *out = $e; |]
    go fun@(Lam Empty f) prelude = do
      t ← compTypeF (argProxy fun)
      v ← varExp <$> freshId
      C.Var n _ ← compExp v
      go (f v) $ prelude >> addParam [cparam| $ty:t $id:n |]
    go fun@(Lam n@(Native l) f) prelude = do
      t ← compTypeF (elemProxy n fun)
      i ← freshId
      let w = varExp i
      C.Var (C.Id m _) _ ← compExp w
      let n = m ++ "_buf"
      withAlias i ('&':m) $ go (f w) $ do
        prelude
        len ← compExp l
        addLocal [cdecl| struct array $id:m = { .buffer = $id:n
                                              , .length=$len
                                              , .elemSize=sizeof($ty:t)
                                              , .bytes=sizeof($ty:t)*$len
                                              }; |]
        addParam [cparam| $ty:t * $id:n |]
    go fun@(Lam (Named s) f) prelude = do
      t ← compTypeF (argProxy fun)
      i ← freshId
      withAlias i s $ go (f $ varExp i) $
        prelude >> addParam [cparam| $ty:t $id:s |]

    argProxy :: Signature (b → c) → Proxy b
    argProxy _ = Proxy

    elemProxy :: Ann [b] → Signature ([b] → c) → Proxy b
    elemProxy _ _ = Proxy
```

Listing 1.4. Signature translation

Apart from allocating a fresh parameter, it creates a local `struct array` object (lines 33–37) on the function stack and initializes it with the length `l` and the buffer parameter. Then compilation continues with `f` applied to the address of the local `struct array` object.

4 Related Work

The purpose of the `Signature` language is to customize the compilation of embedded languages. It is related to Foreign Function Interfaces (FFI) which exist in many forms [7,8]. With the FFI, the signature can be controlled by annotations (e.g. newtype argument wrappers), but annotations are typically limited to individual arguments. The `Signature` language takes the annotations further by allowing them to for example change the arity and the order of the arguments.

MATLAB Coder [9][5] is a tool that generates standalone C and C++ code from MATLAB code. One purpose of MATLAB Coder is to export MATLAB functions to an external system. Since MATLAB is dynamically typed, the same function can operate on values of different type. When generating C code, the user must specify a type for the function, and optionally sizes or size bounds for matrix arguments. This can be done on the command line using what can be seen as a restricted DSL.

However, judging from code examples provided by MathWorks, the signature mapping of MATLAB Coder appears to be rather restricted. For example, stack allocated matrices are passed as two arguments: a pointer to a data buffer and a length vector. If a static size is given for the matrix, the length vector goes away. But if a different argument order is needed, or if one wants to use the same length vector for two different matrices, this likely requires introducing a wrapper function with a different interface.

5 Discussion and Future Work

The `Signature` language enables us to customize the signature of compiled Feldspar functions. It also allows generation of interface code fused with the original function.

Why is a new language needed? Why not just add annotations to the `Lam` abstraction constructor in the Feldspar Core language? Simple annotations, like parameter naming, can be implemented using a combination of newtypes and type classes. In addition to simple annotations, the `Signature` language supports complex manipulations including changing the function arity.

The `Signature` language is a proper extension of the Feldspar Core language, which means it is optional and can co-exist with other extensions. Since the `Signature` is built using a combination of deep and shallow embedding, the language is possible to extend by the end user. Also, the `Signature` language can be seen as a replacement for the top-level lambda abstractions in the Feldspar expression.

[5] Matlab Coder http://www.mathworks.com/products/matlab-coder.

```
-- | Signature annotations
data Ann expr a where
  Empty  :: Ann expr a
  Named  :: String → Ann expr a
  Native :: (VarPred expr a) => expr len → Ann expr [a]

-- | Signatures
data Signature expr a where
  Ret    :: (VarPred expr a) => String → expr a → Signature expr a
  Ptr    :: (VarPred expr a) => String → expr a → Signature expr a
  Lam    :: (VarPred expr a) => Ann expr a → (expr a → Signature expr b)
         → Signature expr (a → b)
```

Listing 1.5. Generalized implementation

A generalized implementation of the `Signature` language is available in the `imperative-edsl` package. That implementation works with any expression language that supports the interface in Listing 1.5. The `imperative-edsl` repository[6] contains an example with a different expression language.

Acknowledgements. This research is funded by the Swedish Foundation for Strategic Research (which funds the Resource Aware Functional Programming (RAW FP) Project) and the Swedish Research Council.

References

1. Axelsson, E., Dévai, G., Horváth, Z., Keijzer, K., Lyckegård, B., Persson, A., Sheeran, M., Svenningsson, J., Vajda, A.: Feldspar: a domain specific language for digital signal processing algorithms. In: Formal Methods and Models for Codesign, MemoCode. IEEE Computer Society (2010)
2. Axelsson, E., Claessen, K., Sheeran, M., Svenningsson, J., Engdal, D., Persson, A.: The design and implementation of Feldspar. In: Hage, J., Morazán, M.T. (eds.) IFL 2010. LNCS, vol. 6647, pp. 121–136. Springer, Heidelberg (2011)
3. Persson, A.: Towards a functional programming language for baseband signal processing. Thesis for the Degree of Licentiate of Engineering (2014). ISSN: 1652–876X
4. Svenningsson, J., Axelsson, E.: Combining deep and shallow embedding for EDSL. In: Loidl, H.-W., Peña, R. (eds.) TFP 2012. LNCS, vol. 7829, pp. 21–36. Springer, Heidelberg (2013)
5. Pfenning, F., Elliot, C.: Higher-order abstract syntax. In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, pp. 199–208. ACM, New York (1988)
6. Mainland, G.: Why it's nice to be quoted: quasiquoting for Haskell. In: Proceedings of the ACM SIGPLAN Workshop on Haskell, pp. 73–82. ACM, New York (2007)

[6] https://github.com/emilaxelsson/imperative-edsl/blob/signatures-camera-ready/examples/Signature.hs.

7. Chakravarty, M.M.: The Haskell 98 Foreign Function Interface 1.0 - An Addendum to the Haskell 98 Report (2003)
8. Chakravarty, M.M.: Foreign inline code: systems demonstration. In: Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell, pp. 119–120. ACM (2014)
9. MathWorks: MATLAB User's Guide (MATLAB Coder), Version: 2011a. http://www.mathworks.com/help/toolbox/coder/index.html

Termination Proofs for Recursive Functions in FoCaLiZe

Catherine Dubois[1] and François Pessaux[2(✉)]

[1] ENSIIE – CEDRIC, Paris, France
catherine.dubois@ensiie.fr
[2] ENSTA ParisTech – U2IS, Paris, France
francois.pessaux@ensta.fr

Abstract. FoCaLiZe is a development environment allowing the writing of specifications, implementations and correctness proofs. It generates both OCaml (executable) and Coq code (for verification needs). This paper extends the language and the compiler to handle termination proofs relying on well-founded relations or measures. We propose an approach where the user's burden is lightened as much as possible, leaving glue code to the compiler. Proofs are written using the declarative proof language provided by FoCaLiZe, and the automatic theorem prover Zenon. When compiling to Coq we rely on the Coq construct `Function`.

Keywords: Formal proof · Functional programming · FoCaLiZe · Coq · Recursion · Termination

1 Introduction

The FoCaLiZe environment [1] (formerly FoCaL [12]) allows one to incrementally build programs or library components with a high level of confidence and quality. FoCaLiZe units may contain specifications, implementations and proofs that the implementations satisfy their specifications. These ones are first-order like formulas while implementations are given as a set of functions in a syntax close to OCaml's one. Proofs are done using hints to the automatic prover Zenon [2,7] in a declarative style. Inheritance and parametrization allow the programmer to reuse specifications, implementations and proofs. FoCaLiZe units are translated into OCaml executable code and verified by the Coq proof assistant.

When specifying properties that the result of a function should follow, we assume that the function does compute a result. This hypothesis is trivial for functions such as identity or square; others may require the programmer to restrict their domain (e.g. division) or lead to extra proof obligations to show that the -recursive- functions terminate. The problem of termination is known as undecidable, however it is tractable in many cases. We rely on classical techniques also used in PVS, Coq or Isabelle consisting in showing that the arguments of each recursive call in the function are strictly lower than the arguments of the initial call according to a measure or a well-founded relation. Some tools,

M. Serrano and J. Hage (Eds.): TFP 2015, LNCS 9547, pp. 136–156, 2016.
DOI: 10.1007/978-3-319-39110-6_8

e.g. Isabelle and Agda [3,10], try to automatically find a convenient lexicographic order and verify termination. In FoCaLiZe, we adopt for termination checking, a solution in line with the general proof discipline which consists in guiding Zenon in its proof search by giving some hints. So, the programmer will indicate the well-founded relation or the measure the proof will use, the recursive argument and provide the proof that the argument is decreasing and the proof that the relation is a well-founded one when necessary. FoCaLiZe provides some helps and computes the statements of the required proofs. Furthermore we do want as much as possible to write the proofs with Zenon. However Zenon relies on first-order and thus cannot cope with higher-order statements, such as the ones that could be required to prove that a certain relation is well-founded. However in many practical cases, the relation is a usual one (e.g. the usual order on natural numbers) or a lexicographic one obtained by combining some standard orders. Hence, with a toolbox offering some standard orders, Zenon will be able to perform the required proofs. As said previously, all the proofs must be checked by Coq. In this context, the FoCaLiZe compiler translates a FoCaLiZe function into a Coq function that is required to be total. Thus, when the recursion is structural, the function is translated into a Coq `Fixpoint`, and we benefit from the syntactic termination verification made by Coq. When the function is recursive but implements a general recursion, we translate it into a general recursive Coq function, using the `Function` construct [5]. This latter requires to determine the relation and asks for proofs that the argument is decreasing and when necessary a proof that the relation is well-founded. In order to be as general as possible, we rely on a compilation scheme that restricts `Function` to use a well-founded relation or a measure defined on the tuple of all the arguments of the recursive function. This general compilation process will ease future work, e.g. allowing the user to set a measure involving several arguments or to make several arguments decrease. Thus, the translation is not syntactic and the compiler has to re-build the relation and the proofs required by `Function` and Coq from the ones provided by the FoCaLiZe programmer. Our approach strongly distinguishes these two views: the user/programmer view and the internal view. The compiler does the glue because it is not the burden of the programmer to fit to a scheme imposed by the certification process (Coq verification). Furthermore, we believe that the user view allows for targeting different compilation schemes or certification environments (e.g. Isabelle).

The rest of the paper is organized as follows. Section 2 presents very briefly the FoCaLiZe environment, in particular its proof language. Section 3 is devoted to the definition of recursive functions whose termination proof requires a well-founded relation: both the user view and the internal view are illustrated on an example. Section 4 follows the same roadmap but for functions that can rely on a measure to prove termination. Section 5 explains the current limitations and proposes some work in progress and perpectives. Many works exist on termination proof, so in Sect. 6, we discuss some related work.

2 Overview of FoCaLiZe

FoCaLiZe [1] is a development environment providing a unique language to write properties, functions and proofs, allowing high-level programming constructs like inheritance, late-binding, and parametrization. It is the continuation of FoC [8] and FoCaL [18].

A FoCaLiZe development is compiled into an executable (or object to link) OCaml code and a Coq term. The OCaml code only contains the computational aspects of the development, while the Coq code is a complete model, also embedding the logical aspects (i.e. properties and proofs). During the compilation process, the logical model is sent to Coq that acts as an assessor (i.e. it checks the code issued by the FoCaLiZe compiler and Zenon).

The basic brick of a FoCaLiZe development is the *species*, a grouping structure embedding *methods* which may be an internal datatype, *properties* (to be proved later), *theorems*, *signatures* (declarations of functions to be defined later) or *definitions*. Once a species has all its signatures defined and properties proved, it can be submitted to an abstraction process turning it into an abstract datatype (a *collection* in the FoCaLiZe terminology) only showing its signatures and properties. Collections can then be used to parameterize species, bringing their own material.

The code generation model extensively uses a dependency calculus to handle late-binding and parametrization and λ-lifts both types and methods to allow code consistency and sharing [17,18]. The dependency calculus has been extended to take into account termination proofs which are not different from other proofs. Roughly speaking, a method m depending on the declaration (type) of a method n is said having a *decl-dependency* on n. In the definition of m, n then gets λ-lifted to circumvent its missing definition or final redefinition. If m depends on the definition of n, then it has a *def-dependency*. In this case, no λ-lifting is done, and the real definition of m is used in n. Dependencies on species parameters methods exist and can be considered as decl-dependencies.

Proofs are written in the FoCaLiZe Proof Language, providing a hierarchical decomposition into intermediate steps [16]. Each step states hypotheses, one goal and a proof of this latter. Each proof can either invoke Zenon to unfold definitions, use previous outer steps, properties, induction or can be a sub-proof.

As an example, the following proof has two outer steps `<1>1` and `<1>2`. The step `<1>1` introduces the hypotheses `h1`, `h2`, `h3` and the sub-goal `c`. It is proved by a 2-step subproof. The step `<2>1` uses `h1` and `h2` to prove `b`. The step `<2>2` uses `<2>1` and `h3` in order to prove `c`. The step `<1>2` ends the whole proof.

```
theorem t : all a b c : bool, a -> (a -> b) -> (b -> c) -> c
proof =
  <1>1 assume a b c : bool,
       hypothesis h1: a, hypothesis h2: a -> b, hypothesis h3: b -> c,
       prove c
    <2>1 prove b by hypothesis h1, h2
    <2>2 qed by step <2>1 hypothesis h3
  <1>2 qed by step <1>1
```

During the compilation process, proofs are compiled and sent to Zenon which tries to find a proof and returns a Coq term. This proof term is then injected in the final generated Coq script which is sent to Coq. If Zenon fails finding a proof with the hints given by the user, then the compilation process fails. Because Zenon does not support higher-order, λ-lifting must be temporarily replaced by Coq `Hypothesis` and `Variable` in `Section`s. This requires the compiler to prepare a suitable Coq environment to host the term that Zenon will return. The compiler must also transmit the user's hints to Zenon. This leads to a slightly verbose generated code for proofs but this remains still readable.

A "backdoor" mechanism is however available, allowing to directly inline Coq scripts in proofs when Zenon does not suffice (e.g. higher-order) or to bind already existing Coq notions. With this solution, the user is required to have a good knowledge of Coq, of the compiler transformations. Unfortunately this mechanism makes the proofs not portable. So it is mostly reserved for the standard library.

3 Well-Founded Relations

The essence of terminating recursion is that there are no infinite chains of nested recursive calls. This intuition is commonly mapped to the mathematical idea of a well-founded relation and we stick to this view which is also the Coq approach. More precisely, Coq uses accessibility to define well-founded relations. Accessibility describes those elements from which one cannot start an infinite descending chain. A relation on T is well-founded when all the elements of type T are accessible (see Fig. 1 for the Coq definition).

In this section we illustrate our approach with the simple example of the function `div` that computes the quotient in the Euclidean division of two positive integers. The function, whose definition is given below, is made total in order to only focus on its termination.

```
let rec div (a, b) =
  if a <= 0 || b <= 0 then 0
  else ( if (a < b) then 0 else 1 + div ((a - b), b))
termination proof = order pos_int_order on a ... ;;
```

3.1 User View

From the user's point of view, despite `div` has two arguments, only the first one `a` is of interest for termination. The well-founded relation used here, `pos_int_order` (of type `int` → `int` → `bool`), is the usual ordering on positive integers provided by the standard library:

```
let pos_int_order (i1, i2)=(0 <=i2)&& (i1 < i2)
```

The well-foundedness obligation of this relation is stated by `is_well_founded pos_int_order`, which relies on the FoCaLiZe standard library's definition of `is_well_founded` as:

```
(fun f => well_founded (fun x y => Is_true (f x y)))
```

The well-foundedness obligation is easily proved thanks to the library theorem `pos_int_order_wf`. Notice that `well_founded` here is the Coq predicate, defined in the Coq standard library (Fig. 1). This exemplifies that a FoCaLiZe specification can mix definitions and properties defined with the FoCaLiZe language together with Coq exported definitions and theorems (and also OCaml definitions but it is not the case in this work).

```
Variable A : Type.
Variable R : A -> A -> Prop.
Inductive Acc (x: A) : Prop :=
     Acc_intro : (forall y:A, R y x -> Acc y) -> Acc x.
(* A relation is well-founded if every element is accessible. *)
Definition well_founded := forall a:A, Acc a.
```

Fig. 1. Coq definition of well_founded

The function `div` having only one recursive call, the only decreasing proof obligation is:

$$\forall a : int, \forall b : int, \neg(a \leq 0 \vee b \leq 0) \rightarrow \neg(a < b) \rightarrow pos\_int\_order(a - b, a)$$

where the conditions on the execution path leading to the recursion must be accumulated as hypotheses.

The termination proof consists in as many steps as there are recursive calls, each one proving the ordering (according to the relation) of the decreasing argument and the initial one, then one step proving that the termination relation is well-founded and an immutable concluding step telling to the compiler to assemble the previous steps, generate some stub code using a built-in Coq script to close the proof. The complete termination proof for `div` is given in Fig. 2. The statements of proof obligations (lines 3–5 and 21) are indicated by the compiler to the user who has just to *cut and paste* them in the source file. The proof that the argument of the recursive call is smaller than the initial one is quite long because Zenon has no support for arithmetic. However, this could be improved by using Zenon Arith, an extension of Zenon able to handle linear arithmetic [11].

To summarize, from the user's point of view, a recursive function whose termination relies on a well-founded relation is given by the four following elements:

1. the relation,
2. the theorem stating that this relation is well-founded,
3. a theorem for each recursive call, stating that the arguments of the recursive calls are smaller than the initial arguments according to the given relation,
4. the recursive definition with a termination proof of the shape (the order of the obligations does not matter):

```
<1>x proofs of decreasing for each recursive call arguments
     (the same statements as the corresponding theorems in point 3,
      even if it is possible to directly inline the proofs instead)
<1>x+1 proof of the relation being well-founded
     (same statement as in point 2, same remark than in steps <1>x)
<1>x+2 qed coq proof {*wf_qed*}
```

```
let rec div (a, b) = ...
  termination proof = order pos_int_order on a
    <1>1 prove all a : int, all b : int,
      ~ (a <= 0 || b <= 0) ->
      ~ (a < b) -> pos_int_order (a - b, a)
      <2>1 assume  a : int, b : int,
           hypothesis H1: ~ (a <= 0 || b <= 0),
           hypothesis H2: ~ (a < b),
           prove pos_int_order (a - b, a)
           <3>1 prove b <= a
                by property int_not_lt_ge, int_ge_le_swap hypothesis H2
           <3>2 prove 0 <= a
                by property int_not_le_gt, int_ge_le_swap, int_gt_implies_ge
                hypothesis H1
           <3>3 prove 0 < b
                by property int_not_le_gt, int_gt_lt_swap hypothesis H1
           <3>4 prove (a - b) < a
                by step <3>1, <3>2, <3>3 property int_diff_lt
           <3>e qed by step <3>4, <3>2 definition of pos_int_order
      <2>e conclude  (* = qed by all previous steps of this nesting level. *)
    <1>2 prove is_well_founded (pos_int_order)
       by  property pos_int_order_wf
   <1>e qed coq proof {*wf_qed*} ;
```

Fig. 2. Termination proof for `div`

3.2 Internal View

From the compiler's point of view, the code generation is split in 4 steps:

1. **Creation of the relation expected by `Function`.** This one takes two tuples with as many arguments as the user's recursive function has. It extracts the decreasing one from each tuple, and applies the user's relation on them.
2. **Creation of the user-side termination theorem** containing the compiled proof of the user. This theorem only operates on the decreasing argument, hence uses the user's relation. This theorem is the conjunction of the decreasing obligations and the well-foundedness obligation.
3. **Creation of the `Function`-side termination theorem.** This theorem operates on the tuple of arguments of the function and uses the generated relation. Roughly speaking, this theorem states the same property as the previous one, but operating on tuples and referring to the relation synthesized at step 1. This proof is fully done by the compiler, using the proof of well-foundedness of the user's relation and a Coq theorem (`wf_inverse_image`) stating that the reverse image of a well-founded relation by any function (here, tuple projectors) is a well-founded relation.
4. **Creation of the recursive function definition using `Function`.** The `Function` body is obtained using the usual FoCaLiZe compilation scheme, the termination part is filled with the relation generated at step 1 and a final proof built with the theorem generated at the previous step.

Figure 3 gives an overview of the structure of the compilation of a function. Grayed blocks are those generated by the compiler while white ones are the code of the user.

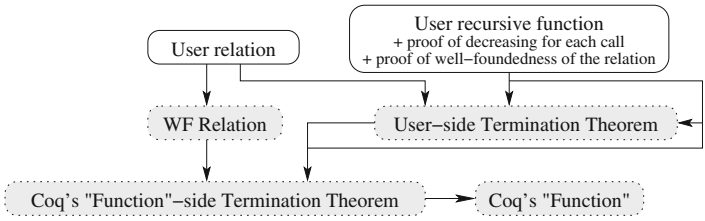

Fig. 3. Global structure of a function with its proof

3.2.1 `Function`

In this part, we briefly recall how the Coq command `Function` works `Function` [5] (see also the Coq manual reference).

The Coq proof assistant provides the command that allows the definition of both structural and well-founded recursive functions. When defining a non-structurally recursive function, the user is asked to provide a well-founded relation (or a measure function) and an argument (the decreasing one) and to show that the corresponding arguments in the recursive calls are smaller than the initial ones according to the well-founded relation (or the measure). Furthermore, the user has also to prove that the relation is indeed a wellfounded one. So a definition using this construct looks like a definition by pattern-matching in a functional language, annotated with a measure or a wellfounded relation and completed with some proof scripts discharging the termination proof obligations generated by `Function`. Note that the statements of the termination proof obligations do not appear in the definition. These ones are made explicit in a FoCaLiZe definition. Once the definition has been accepted by Coq, a set of definitions is automatically derived, in particular a fixpoint equation and an induction principle that follows the structure of the function.

3.2.2 Creation of the Relation Expected by `Function`

The expected relation takes two arguments `__x` and `__y` being tuples of all the function arguments. To extract the one used for decreasing, a built-in projection is used. In Coq, tuples are encoded as pairs nested to the left: $(x, y, z) \equiv ((x, y), z)$. Projections are part of the low-level standard library of FoCaLiZe and internally known by the compiler which generates the right name from the number of components of the tuple and the position of the component to extract. Here the projection `__tpl_firstprj2` extracts the first component of a 2-uple. and is defined as:

```
Definition __tpl_firstprj2 (__var_a : Set) (__var_b : Set)
  (x : (__var_a * __var_b)) : __var_a := __left __var_a __var_b x.
```

Once components are extracted, there remains to apply the user relation to them. The corresponding Coq definition is as follows (where _ stands for inferred arguments):

```
Definition div_wforder (__x __y : (int__t * int__t)) : Prop :=
  Is_true
    (pos_int_order (__tpl_firstprj2 _ _ __x) (__tpl_firstprj2 _ _ __y)).
```

3.2.3 Creation of the User-Side Termination Theorem

The user's termination proof must now be compiled and generated. Except the concluding step, all steps are compiled using the usual FoCaLiZe compilation process. Because Zenon does not handle higher-order, dependencies cannot be λ-lifted (see Sect. 2). Instead, we enclose the proof in a Coq section, where dependencies lead to `Variable` and `Hypothesis` clauses. For the same reason, each proof step is also embedded in a `Section`. In the generated proof below, line numbers and proof steps refer to the code given at Fig. 2.

```
Section Proof_of_div.
  Section __A_1.
    (* Step <1>1 line 3. *)
    Theorem __A_1_LEMMA :
      forall a : int__t, forall b : int__t,
        ~ Is_true (_bar__bar_ (_lt__eq_ a 0) (_lt__eq_ b 0)) ->
        ~ Is_true (_lt_ a b) -> Is_true (pos_int_order (_dash_ a b) a).
    (* ... Zenon proof term inlined here ... *)
    End __A_1.
  Section __A_2.
    (* Step <1>2 line 21. *)
    Theorem __A_2_LEMMA : ((is_well_founded _) pos_int_order).
    (* ... Zenon proof term inlined here ... *)
    End __A_2.
  (* Theorem's body. *)
  Theorem for_zenon_div :
    (forall a : int__t, forall b : int__t,
       ~ Is_true (_bar__bar_ (_lt__eq_ a 0) (_lt__eq_ b 0)) ->
       ~ Is_true (_lt_ a b) -> Is_true (pos_int_order (_dash_ a b) a))
      /\
      (well_founded (fun __a1 __a2 => Is_true (pos_int_order __a1 __a2))).
  Proof.
  generalize __A_2_LEMMA.
  generalize __A_1_LEMMA.
  unfold is_well_founded.
  intros.
  SplitandAssumption.
  Qed.
End Proof_of_div.
```

Once each step of the proof but the concluding one has been generated, the compiler detects that the concluding one refers to termination. It then generates and proves the theorem `for_zenon_div` stating the termination obligations as the user sees them: only dealing with the unique decreasing argument. This theorem's statement is the conjunction of all the decreasing lemmas and the well-foundedness of the user's relation introduced in Sect. 3.1. Since the user is expected to have proved the different parts of this conjunction in the previous steps, each lemma introduced by the steps is generalized, and a Coq tactic `SplitandAssumption` is used to automate splitting the goal then picking in the context to solve each sub-goal.

3.2.4 Creation of the Function-Side Termination Theorem

Once the user-side theorem is available, its counterpart as expected by Function must be generated as a separate theorem. This theorem states the same proof obligations, but dealing with the tuple of arguments of the recursive function. Hence it must use the relation automatically generated in Sect. 3.2.2 and the user-side termination theorem generated in Sect. 3.2.3 as shown in the following generated code.

```
Theorem div_termination :
  (forall a : int__t, forall b : int__t,
   ~ Is_true (_bar__bar_ (_lt__eq_ a 0) (_lt__eq_ b 0)) ->
   ~ Is_true (_lt_ a b) -> div_wforder ((_dash_ a b), b) (a, b))
   /\
   (well_founded (div_wforder)).
  Proof.
  unfold div_wforder;simpl.
  elim for_zenon_div.
  intro __user_dec1.
  intro __user_rem_dec_n_wf.
  (* Separate decreasing obligations and well-foundedness. *)
  split.
  auto. (* For each rec call. *)
  (* There, only remains the well-foundedness obligation. *)
  set (R := (fun __a __b => Is_true (pos_int_order __a __b))).
  change
    (well_founded (fun __c __d : (int__t * int__t)
      => R (__tpl_firstprj2 _ _ __c) (__tpl_firstprj2 _ _ __d))).
  apply wf_inverse_image.
  assumption.
  Qed.
```

The proof of the theorem is automatically generated by the compiler. The proof proceeds as follows. It first introduces in the hypothesis context the conclusions of the user-side termination theorem (named here __user_dec1 and __user_rem_dec_n_wf). The proof is split, thus separating the decreasing and well-foundedness obligations. Decreasing obligations are proved automatically thanks to the simplification of the goal and hypotheses found in the context. This is to be repeated as many times as there are recursive calls. The last part proves that the relation defined for Function is well-founded. We reformulate the goal statement such that it appears as well-foundedness of the user-defined relation (denoted by R in the proof script) composed with the projection needed to extract the decreasing argument from the tuple of arguments (here __tpl_firstprj2). The proof ends with the application of the Coq standard library theorem wf_inverse_image that establishes that if a relation is well-founded then this relation composed with any function is also well-founded. It asks for the well-foundedness of the user-relation which is a hypothesis (here __user_rem_dec_n_wf).

3.2.5 Creation of the Recursive Function Definition Using Function

The Coq Function can now be generated, using the generated relation (in our example, div_wforder). The compiler emits the "glue code" of the final proof, using the theorem generated in Sect. 3.2.4 and some low-level theorems of the

FoCaLiZe standard library. These low-level theorems mostly deal with properties about the equality.

```
Function div (__arg: (int__t * int__t))
  {wf div_wforder __arg}: int__t :=
  match __arg with
    | (a, b) =>
       (if (_bar__bar_ (_lt__eq_ a 0) (_lt__eq_ b 0)) then 0
       else ((if ((_lt_ a b)) then 0 else _plus_ 1 (div ((_dash_ a b), b)))))
    end.
  Proof.
    elim div_termination.
    intros __for_function_dec1 __for_function_rem_dec_n_wf.
    split. intros.
    eapply __for_function_dec1 ; eauto ||
      (apply coq_builtins.EqTrue_is_true; assumption) ||
      (apply coq_builtins.IsTrue_eq_false2; assumption) ||
      (apply coq_builtins.syntactic_equal_refl).
    (* Remaining proof of well-foundedness  ... *)
    assumption.
    Qed.
```

Figure 4 gives a detailed structure of the complete compilation of a function `f` whose termination relies on a relation `r`. Grayed blocks are those generated by the compiler while white ones are the code of the user.

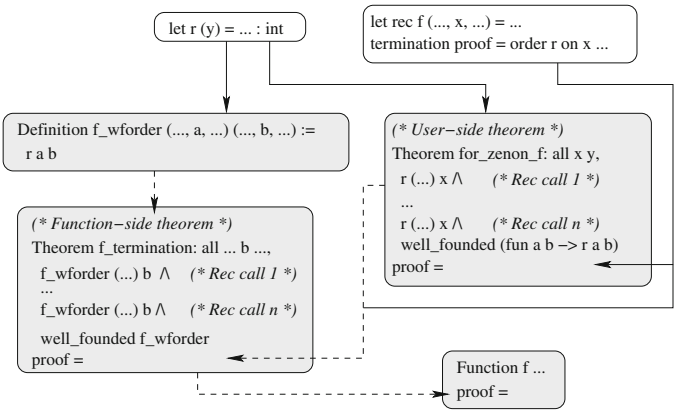

Fig. 4. Detailed structure of a function with its proof

4 Measure Functions

We consider here the particular case when the termination relies on a *measure* – a function that returns a natural number – which must decrease at each recursive call. We want to ease such termination proofs even if it would be possible for the user to use the previous approach, by constructing himself a well-founded relation from the measure. Precisely, the compiler does this job for him.

A measure has to be positive, which will be a proof obligation. However, the relation built from the measure being internalized, its well-foundedness is no more asked of the user. From the user's point of view, the proof obligation for

each recursive call must now show that the measure decreases on the argument of interest between each call. The compiler must build a well-founded relation from the measure, operating on all the arguments of the function, to prove its well-foundedness and build the final proof expected by Coq construct `Function`.

Let us notice that `Function` natively supports a `measure` annotation that we do not use. Indeed, we think that only relying on a well-founded relation leaves the compilation scheme more open to other logical target languages.

We illustrate the approach with the function `qsort` that implements the quicksort algorithm and thus sorts the elements of a list. The type of these elements is provided by the parameter `A` of the species `AList` together with an ordering method `le`. The function `qsort` performs two recursive calls on two sub-lists obtained by splitting the initial list. The termination is ensured by the shorter lengths of these sub-lists compared to the initial list. We first write the method `length` whose termination is simply structural on its argument `l`. The sorting function requires a `partition` method to split a list into the two sub-lists respectively containing the elements lower or equal and strictly greater than a "pivot" value, according to the method `le` of the parameter `A`. Finally, we write the method `qsort`, stating a termination proof using the measure `length` on its argument `l`, but we do not show the proof itself yet:

```
species AList (A is Comparable) =
  let rec length (l : list (A)) =
    match l with
    | [] -> 0
    | h :: q -> 1 + length (q)
    termination proof = structural l ;

  let rec partition (l , x : A) =
    match l with
    | [] -> ([], [])
    | h :: q ->
       let p = partition (q, x) in
       if A!le (h, x) then
         (h :: (fst (p)), snd (p))
       else (fst (p), h :: (snd (p)))
  termination proof = structural l ;

  let rec qsort (l : list (A)) =
    match l with
    | [] -> []
    | x :: r ->
      match r with
      | [] -> l
      | y :: q ->
         let p = partition (r, x) in
         app (qsort (fst (p)),
              x :: (qsort (snd (p))))
  termination proof =
    measure length on l ... ;
end ;;
```

Using the species and collection parameters features of FoCaLiZe, this example also shows that the compilation model introduced in this paper fits the usual dependency calculus and λ-lifting mechanisms used by the compiler.

Note that in this example, `qsort` has only one argument. In this particular case, the mechanism described in Sect. 3 to manage the tuple of all the arguments of a function is irrelevant here.

4.1 User View

From the user's point of view, the measure being `length` on the argument `l`, the first proof obligation is $\forall l : list(A), 0 \leq length(l)$.

Then, the method `qsort` having two recursive calls, the two decreasing proof obligations are:

$$\forall l : list(A), \forall x : A, \forall r : list(A), \forall y : A, \forall q : list(A), \forall p : list(A) * list(A),$$
$$(l = x :: r) \rightarrow (r = y :: q) \rightarrow (partition(r, x) = p) \rightarrow length(fst(p)) < length(l)$$

and

$$\forall l : list(A), \forall x : A, \forall r : list(A), \forall y : A, \forall q : list(A), \forall p : list(A) * list(A),$$
$$(l = x :: r) \rightarrow (r = y :: q) \rightarrow (partition(r, x) = p) \rightarrow length(fst(p)) < length(l)$$

where variables bound on the execution path leading to the recursive call must be accumulated as hypotheses. Here, the recursive calls being in pattern-matching cases, `x` and `r` must be related to the matched value `l` (idem for `y`, `q` and `r`). The core of the decreasing facts is the $<$ relation between the length of the recursive calls arguments and the length of the list in the current call.

For readability, instead of inlining the proofs of these obligations, the user can state two related properties or theorems before the function `qsort` itself:

```
property length_pos : all l: list (A), 0 <= length (l) ;

theorem mes_decr_fst :
  all l : list (A), all x : A, all r : list (A), all y : A,
  all q : list (A), all p : list (A) * list (A),
  (l = x :: r) -> (r = y :: q) -> (partition (r, x) = p) ->
    length (fst (p)) < length (l)
proof = ... ;

theorem mes_decr_snd :
  all l : list (A), all x : A, all r : list (A), all y : A,
  all q : list (A), all p : list (A) * list (A),
  (l = x :: r) -> (r = y :: q) -> (partition (r, x) = p) ->
    length (snd (p)) < length (l)
 proof = ... ;
```

Note that `length_pos` is a *property*, not a theorem: it is not yet proved. Hence, thanks to the dependency calculus, it will be λ-lifted. This shows that our code generation model is not impacted by termination proofs.

Now the termination proof (see Fig. 5) consists in as many steps as there are recursive calls, each one proving the strict decreasing of the measure, then one step proving that the measure is positive and an immutable concluding step telling the compiler to assemble the previous steps and generate some stub code to close the proof.

To summarize, from the user's point of view, a recursive function whose termination relies on a measure is given by the four following items:

1. the measure function returning a regular integer (which raises the issue that $<$ is well-founded on naturals, not integers),
2. the theorem stating that the measure is always positive or null,
3. a theorem for each recursive call, stating that the measure on the argument of interest decreases,
4. the recursive definition with a termination proof of the following shape:

```
<1>x proofs of decreasing for each recursive call
     (the same statements as corresponding theorems in point 3,
      even if it is possible to directly inline the proofs instead)
<1>x+1 proof of the measure being always positive or null
     (same statement as in point 2, same remark than in steps <1>x)
<1>x+2 qed coq proof {*wf_qed*}
```

```
let rec qsort (l : list (A)) = ...
termination proof = measure length on l
  <1>1 prove all l : list (A), all x : A,
       all r : list (A), all y : A, all q : list (A),
       all p : list (A) * list (A),
       (l = x :: r) -> (r = y :: q) -> (partition (r, x) = p) ->
         length (fst (p)) < length (l)
        by property mes_decr_fst
  <1>2 prove all l : list (A), all x : A,
       all r : list (A), all y : A, all q :  list (A),
       all p : list (A) * list (A),
       (l = x :: r) -> (r = y :: q) -> (partition (r, x) = p) ->
         length (snd (p)) < length (l)
       by property mes_decr_snd
  <1>3 prove all l: list (A), 0 <= length (l)
       by property length_pos
  <1>e qed coq proof {*wf_qed*} ;
```

Fig. 5. Termination proof for `qsort`

4.2 Internal View

From the compiler's point of view, the code generation is split in 4 parts:

1. **Creation of the relation expected by `Function`.** It takes two tuples with as many arguments as the user's recursive function has. It extracts the decreasing one from each tuple, say x and y. It finally states that the measure on y is positive and that the measure on x is strictly lower than the measure on y. The first part of this definition is needed by the proof done in point 3.
2. **Creation of the user-side termination theorem** containing the compiled proof of the user. This theorem only operates on the decreasing argument, hence uses the user's measure function. This theorem is the conjunction of the decreasing obligations and the measure positive obligation.
3. **Creation of the `Function`-side termination theorem**. This theorem operates on the tuple of arguments of the function and uses the generated relation. Roughly speaking, this theorem states the same property as the previous one, but operating on tuples and referring to the relation generated at step 1. This proof is fully done by the compiler, using the user's proof that the measure is always positive and a Coq theorem stating that the usual order on positive integers is well-founded.
4. **Creation of the recursive function definition using `Function`.** This step computes the `Function` body and the final proof built with the theorem generated at the previous step for the termination part.

The last point is exactly the same as for a termination proof using a well-founded relation. This allows a code generation model as close as possible for both kinds of proofs.

Figure 6 gives an overview of the compilation process of a function. Grayed blocks are those generated by the compiler while white ones are the code of the user.

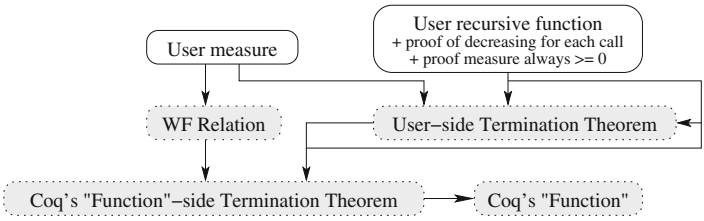

Fig. 6. Global structure of a function with its proof

4.2.1 Creation of the Relation Expected by `Function`

The dependency calculus of FoCaLiZe, extended to termination proofs indicates that the proof decl-depends on the declarations of `length`, `app`, `partition`, `length_pos`, `mes_decr_fst` and `mes_decr_snd`. Hence these dependencies are first λ-lifted and lead to extra parameters (whose names are prefixed by `abst_`) for the relation. Since Coq makes polymorphism explicit, the types of collection parameters also must be λ-lifted (here `_p_A_T: Set` for the parameter `A`).

The expected relation takes two arguments `__x` and `__y` on which the user measure is applied. There remains to compare the results with the standard order on naturals ($<$, which is generated as `_lt_`, itself bound to the corresponding Coq definition). The generated relation is given below (where `_amper__amper_` denotes the conjunction):

```
Definition qsort_wforder (_p_A_T : Set)
  (abst_app : list__t _p_A_T -> list__t _p_A_T -> list__t _p_A_T)
  (abst_length : list__t _p_A_T -> int__t)
  (abst_partition : list__t _p_A_T -> _p_A_T ->
                    list__t _p_A_T * list__t _p_A_T)
  (abst_length_pos :
    forall l : list__t _p_A_T, Is_true (_lt__eq_ 0 (abst_length l)))
  (abst_mes_decr_fst : ...) (abst_mes_decr_snd : ...) :=
  (__x __y : list__t _p_A_T) : Prop :=
    Is_true (_amper__amper_
              (_lt__eq_ 0 (abst_length __y))
              (_lt_ (abst_length __x) (abst_length __y))).
```

4.2.2 Creation of the User-Side Termination Theorem

Next, the user termination proof must be generated. We find again the `Section` mechanism used in Sect. 3.2.3 in order to keep Zenon in a first-order environment. The shape of the generated code is very close to the one generated for termination proofs by a relation and is shown below. Line numbers and proof steps refer to the code given at Fig. 5.

```
Section qsort.
  Section Proof_of_qsort.
  Variable _p_A_T, abst_app, abst_length, abst_partition : ...
  Hypothesis abst_length_pos, abst_length_pos, abst_mes_decr_fst,
       abst_mes_decr_snd : ...
  Section __E_1.
    (* Step <1>1 line 3. *)
```

```
    Theorem __E_1_LEMMA : forall ...,
      Is_true ((_eq_ _) l (@ List.cons _p_A_T x r)) ->
      Is_true ((_eq_ _) r (@ List.cons _p_A_T y q)) ->
      Is_true ((_eq_ _) (abst_partition r x) p) ->
      Is_true (_lt_ (abst_length ((fst _ _) p)) (abst_length l)).
    (* ... Zenon proof term inlined here ... *)
    End __E_1.
  Section __E_2.
  (* Step <1>2 line 9. *)
  Theorem __E_2_LEMMA : forall ...,
    Is_true ((_eq_ _) l (@ List.cons _p_A_T x r)) ->
    Is_true ((_eq_ _) r (@ List.cons _p_A_T y q)) ->
    Is_true ((_eq_ _) (abst_partition r x) p) ->
    Is_true (_lt_ (abst_length ((snd _ _) p)) (abst_length l)).
    (* ... Zenon proof term inlined here ... *)
    End __E_2.
  Section __E_3.
  (* Step <1>3 line 16. *)
  Theorem __E_3_LEMMA :
    forall l : (list__t _p_A_T), Is_true (_lt__eq_ 0 (abst_length l)).
    (* ... Zenon proof term inlined here ... *)
  (* Theorem's body. *)
  Theorem for_zenon_qsort :
    (forall ...,
      (Is_true ((_eq_ _) l (@ List.cons _p_A_T x r))) ->
      (Is_true ((_eq_ _) r (@ List.cons _p_A_T y q))) ->
      (Is_true ((_eq_ _) (abst_partition r x) p)) ->
      Is_true (_lt_ (abst_length ((snd _ _) p)) (abst_length l)))
    /\
    (forall ...,
      (Is_true ((_eq_ _) l (@ List.cons _p_A_T x r))) ->
      (Is_true ((_eq_ _) r (@ List.cons _p_A_T y q))) ->
      (Is_true ((_eq_ _) (abst_partition r x) p)) ->
      Is_true (_lt_ (abst_length ((fst _ _) p)) (abst_length l)))
    /\
    (forall __x, Is_true (_lt__eq_ 0 (abst_length __x))).
  Proof.
  generalize __E_3_LEMMA. generalize __E_2_LEMMA. generalize __E_1_LEMMA.
  unfold is_well_founded. intros. SplitandAssumption. Qed.
End Proof_of_qsort.
```

Once each step of the proof but the concluding one has been generated, the compiler detects that the concluding one refers to termination. It then generates and proves the theorem `for_zenon_qsort` stating the termination obligations as the user sees them. The statement of this theorem is the conjunction of all the decreasing lemmas and the positiveness of the measure introduced in Sect. 4.1. The proof generated by the compiler is again very close to the one for a termination by a well-founded relation, and uses the same automation techniques.

4.2.3 Creation of the `Function`-Side Termination Theorem

Since the user-side theorem is available, its counterpart as expected by `Function` has to be emitted. Like for proofs by a relation in Sect. 3.2.4, this theorem states the same proof obligations as the user-side theorem but dealing with the well-founded relation previously built. This theorem requires the same `Variable` and `Hypothesis` as the user-side one since the dependencies are the same. To lighten the presentation we do not repeat them in the following generated code sample.

```
Theorem qsort_termination :
  (forall ...,
    (Is_true ((_eq_ _) l (@ List.cons _p_A_T x r))) ->
    (Is_true ((_eq_ _) r (@ List.cons _p_A_T y q))) ->
    (Is_true ((_eq_ _) (abst_partition r x) p)) ->
    (qsort_wforder
      _p_A_T abst_app abst_length abst_partition abst_length_pos
      abst_mes_decr_fst abst_mes_decr_snd) ((snd _ _) p) l)
   /\
   (forall ...,
     (Is_true ((_eq_ _) l ((@ List.cons _p_A_T x r)))) ->
     (Is_true ((_eq_ _) r ((@ List.cons _p_A_T y q)))) ->
     (Is_true ((_eq_ _) (abst_partition r x) p)) ->
     (qsort_w forder
       _p_A_T abst_app abst_length abst_partition abst_length_pos
       abst_mes_decr_fst abst_mes_decr_snd) ((fst _ _) p) l)
   /\
   (well_founded
     (qsort_w forder
       _p_A_T abst_app abst_length abst_partition abst_length_pos
       abst_mes_decr_fst abst_mes_decr_snd)).
Proof.
unfold qsort_wforder;simpl.
elim (for_zenon_qsort _p_A_T abst_app abst_length abst_partition
        abst_length_pos abst_mes_decr_fst abst_mes_decr_snd).
intro __user_dec1.
intro __user_rem_dec_n_wf.
(* Separate decreasing obligations and well-foundation. *)
split. intros. apply coq_builtins.andb_intro; eauto.
split. intros. apply coq_builtins.andb_intro; eauto.
(* There, only remains the well-foundation obligation. *)
set (R := fun x y : int__t =>
            Is_true (_amper__amper_ (_lt__eq_ 0 y) (_lt_ x y))).
change (well_founded (fun __c __d : ((list__t _p_A_T)) =>
                        R (abst_length __c) (abst_length __d))).
apply wf_inverse_image.
apply wf_incl with (R2 := (fun x y : Z => 0 <= y /\ x < y)).
unfold inclusion, R.
unfold int__t, _amper__amper_, _lt__eq_, _lt_, bi__and_b, bi__int_leq,
     bi__int_lt.
intros x y.
elim (Z_le_dec 0 y); intro; elim (Z_lt_dec x y); simpl; intros;
intuition. apply (Zwf_well_founded 0). Qed.
```

Again, the proof of the theorem is automatically generated by the compiler. It also consists in separating the decreasing and well-foundedness obligations. The interconnecting Coq script is more complex than for proofs by a relation. In particular, in the first part about measure decreasing, the proof is not so straightforward than previously because we have to deal with integers and the $<$ relation. The well-foundedness obligation proof follows the same scheme, it relies on three library theorems, `wf_inverse_image` to deal with the relevant projection, `Zwf_well_founded 0` which is the proof that the restriction of $<$ to the positive integers is a well-founded relation and `wf_incl` establishing that a relation included in a well-founded one, is also well-founded.

4.2.4 Creation of the Recursive Function Definition Using `Function`

The Coq `Function` can now be generated, using the generated relation. Exactly in the same way as for proofs with a relation, the compiler ensures the glue to build the final proof, taking benefit from the theorem generated in Sect. 4.2.3.

In particular, the final soldering Coq script is exactly the same as for a termination proof with a relation. Because of dependencies in the user code that were λ-lifted, `qsort_wforder` had some extra arguments. They must be instantiated to provide `Function` with a correct relation in its `wf` clause. This is done by applying `qsort_wforder` to the effective methods definitions computed by the late-binding resolution (`_p_A_T`, `abst_length`, etc.). Below is shown the final and complete code of the compiled function `qsort`.

```
Function qsort (__arg: (list__t _p_A_T))
  {wf (qsort_wforder _p_A_T abst_app abst_length abst_partition
         abst_length_pos abst_mes_decr_fst abst_mes_decr_snd) __arg}:
  list__t _p_A_T :=
  match __arg with (l) =>
    match l with
     | List.nil => @ List.nil  _p_A_T
     | List.cons x r =>
         match r with
          | List.nil => l
          | List.cons  y q =>
              let p := abst_partition r x in
              abst_app (qsort ((fst _ _) p))
                       (@ List.cons _p_A_T x ((qsort ((snd _ _) p))))
          end
     end
end.
Proof.
elim qsort_termination.
intros __for_function_dec1 __for_function_rem_dec_n_wf.
elim __for_function_rem_dec_n_wf. clear __for_function_rem_dec_n_wf.
intros __for_function_dec2 __for_function_rem_dec_n_wf.
split. intros. eapply __for_function_dec1 ; eauto || (apply coq_builtins.
    EqTrue_is_true; assumption) || (apply coq_builtins.IsTrue_eq_false2;
    assumption) || (apply coq_builtins.syntactic_equal_refl).
split.intros. eapply __for_function_dec2 ; eauto || ... (* Same than above.
    *)
(* Remaining well-foundation... *)
assumption. Qed.
```

Figure 7 gives a detailed structure of the complete compilation of a function `f` whose termination relies on a measure `m`. Grayed blocks are those generated by the compiler while white ones are the code of the user.

5 Limitations and Perspectives

The termination proofs as described in this paper are available in the FoCaLiZe repository. A "toolbox" containing some low-level theorems proved in Coq is also available to "manually" wrap a measure in a well-founded relation.

Termination proofs using lexicographic orders are currently under study. Again, we want to stick to our approach that provides the user with some comfort for termination proofs. The compiler will have to generate itself the lexicographic order and its well-foundedness proof from the user's orders and their own well-foundedness proofs.

Some known limitations exist. Termination proofs being based on the Coq construct `Function`, only methods of species and toplevel functions are supported. Local recursive functions cannot be handled this way. Nested recursion

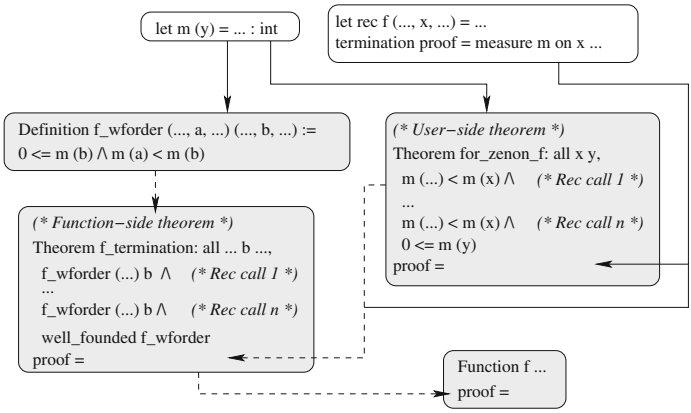

Fig. 7. Detailed structure of a function with its proof

is also not supported: the construct `Function` does also not. Mutual recursive functions cannot be compiled with the present scheme. Although encodings exist to deal with such functions, several issues already appear: what are the proof obligations to impose to the user, how strong will they be impacted by the encoding, how understandable will these obligations become, how to mix several termination proof schemes?

Aside the kinds of termination proofs, it is not fully clear how to provide late-binding at the termination level. In FoCaLiZe, definitions can use methods only declared. The dependency calculus allows λ-lifting late-bound symbols. The termination proof of a recursive function is part of its definition, hence delaying the proof means delaying the function definition. The simplest solution would be to consider the function as a simple signature until its proof is provided. This would however delay other proofs depending on the definition of the function, despite the termination proof is not relevant for them. Indeed, either the function terminates and other functions are not interested in its termination, or it does not terminate, and the complete logical model is possibly inconsistent.

Finally, only one of the function's parameters can be used to prove decreasing. There is no particular obstacle for this extension, it is only an implementation matter. This restriction only impacts proofs by a measure (which could use several parameters), since such an extension for orders directly leads to lexicographic orders.

6 Related Work

Our work is in line with those about the definition of recursive functions in theorem provers, and more precisely the proposals made to facilitate the definition and reasoning with general recursive functions. All these propositions allow some separation of the computational and logical parts, as we do. As said previously, we would like to go a step further in this direction and defer a termination proof.

TFL [19], implemented for both HOL4 and Isabelle, allows the definition and reasoning about total recursive programs written in a purely functional manner. In this context, establishing the termination of a function requires the introduction of a well-founded relation (proved as such) and the proof that the arguments of the recursive calls decrease according to this relation.

Coq provides the `Function` package [5] that allows one to define recursive functions in a way close to TFL. It relies on previous work done on termination by Balaa and Bertot [4]. The main strength concerns induction principles automatically generated from the algorithmic definition of the function, e.g. functional induction. We decided to compile our - non structural - recursive functions to Coq functions defined with `Function` because of traceability. Except for the usual modifications due to the compilation of dependencies and the proof obligations part, the text of the FoCaLiZe recursive function and the Coq one are very close. Furthermore the fixpoint equation generated by `Function` and by the FoCaLiZe compiler for Zenon reasoning are again very close. Another compilation choice would have been to bypass `Function` and its limitations altogether and generate Coq definitions on top of the basic Coq `Wf` package that provides a well-founded induction principle. We could also have used Bertot and Komendantsky's approach [6] to general recursion. As said previously, the shapes of both definitions in Coq and FoCaLiZe would have been too different and furthermore it would have been more difficult to generate the proofs.

In [15], Krauss provides a way to define general well-founded recursion in Isabelle. It is based on principles close to those used by `Function` in Coq, and goes further in some directions (nested recursion, mutual recursion and partiality). The main strength of this work is the advances in the automation of termination proofs. It can prove automatically termination of a certain class of functions by searching for a suitable lexicographic combination of size measures [10]. The termination for another class is handled by using the Size-Change principle [14]. This principle is also used in [13] to provide a tool that automatically determines whether one or mutually recursive functions terminate. This approach allows for a local increasing of recursive arguments. FoCaLiZe does not intend to automatically find proofs: it lets this task to an external prover thanks to some hints given by the user. Moreover, it is yet unclear how to generate a Coq term from such a termination proof.

More generally, Bove, Krauss and Sozeau review in [9] different techniques that have been proposed to formalize partial and general recursive functions in interactive theorem provers.

7 Conclusion

This work integrates in FoCaLiZe means to prove the termination of recursive functions that are not only structural. It brings the ability to state a well-founded relation or a measure and write the termination proof using the usual FoCaLiZe proofs shape: with a hierarchical structure and using the Zenon automated theorem prover to relieve the user. Proof obligations are indicated to the user

by the compiler, which avoids tedious errors and the need to guess what proof obligations the compiler is expecting. The proofs done by the user are based on the usual termination proof obligations for a well-founded relation or a measure, and ask the user only to consider the decreasing argument of his function. This point of view is indeed the one always used for handmade proofs and it would be annoying to ask the user to cope with all the other arguments since they are of no interest for the termination.

Termination proofs can transparently involve late-bound methods (i.e. methods that are only declared, e.g. properties used in proof obligations or even the measure or the well-founded relation) thanks to the λ-lifting mechanism used by FoCaLiZe.

A more general compilation scheme seems required to solve the pending issues and have a more unified code generation model. But this scheme remains to be found. However, the current work is already an appreciable help for the user and a first step toward a more global problem. It already allowed to write some previously assumed termination proofs of the FoCaLiZe standard library.

Acknowledgements. We thank Renaud Rioboo for the useful discussions and case studies. Thanks to Julien Forest for the helpful discussions about `Function`. Lastly we thank William Bartlett for his work on a very first prototype.

References

1. http://focalize.inria.fr/
2. http://zenon-prover.org/
3. Abel, A., Altenkirch, T.: A predicative analysis of structural recursion. J. Funct. Program. **12**(1), 1–41 (2002)
4. Balaa, A., Bertot, Y.: Fix-point equations for well-founded recursion in type theory. In: Aagaard, M.D., Harrison, J. (eds.) TPHOLs 2000. LNCS, vol. 1869, pp. 1–16. Springer, Heidelberg (2000)
5. Barthe, G., Forest, J., Pichardie, D., Rusu, V.: Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant. In: Hagiya, M. (ed.) FLOPS 2006. LNCS, vol. 3945, pp. 114–129. Springer, Heidelberg (2006)
6. Bertot, Y., Komendantsky, V.: Fixed point semantics and partial recursion in Coq. In Proceedings of PPDP 2008, Valencia, Spain, pp. 89–96 (2008)
7. Bonichon, R., Delahaye, D., Doligez, D.: Zenon: an extensible automated theorem prover producing checkable proofs. In: Dershowitz, N., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. LNCS, vol. 4790, pp. 151–165. Springer, Heidelberg (2007)
8. Boulmé, S., Hardin, T., Hirschkoff, D., Ménissier-Morain, V., Rioboo, R.: On the way to certify computer algebra systems. In: Proceedings of the Calculemus Workshop of FLOC 1999. ENTCS, vol. 23. Elsevier (1999)
9. Bove, A., Krauss, A., Sozeau, M.: Partiality and recursion in interactive theorem provers an overview. Math. Struct. Comput. Sci. FirstView, 1–51 (2015)
10. Bulwahn, L., Krauss, A., Nipkow, T.: Finding lexicographic orders for termination proofs in Isabelle/HOL. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 38–53. Springer, Heidelberg (2007)

11. Bury, G., Delahaye, D.: Integrating simplex with tableaux. In: Nivelle, H.D. (ed.) Automated Reasoning with Analytic Tableaux and Related Methods. LNCS, vol. 9323, pp. 86–101. Springer, Heidelberg (2015)
12. Dubois, C., Hardin, T., Donzeau-Gouge, V.: Building certified components within FOCAL. Trends in Functional Programming, vol. 5, pp. 33–48 (2006)
13. Hyvernat, P.: The Size-change termination principle for constructor based languages. Logical Methods Comput. Sci. **10**(1) (2014)
14. Krauss, A.: Certified size-change termination. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 460–475. Springer, Heidelberg (2007)
15. Krauss, A.: Partial and nested recursive function definitions in higher-order logic. J. Autom. Reason. **44**(4), 303–336 (2010)
16. Lamport, L.: How to write a proof. Research report, Digital Equipment Corporation (1993)
17. Pessaux, F.: Focalize: Inside an F-IDE. In: Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014. EPTCS, vol. 149, pp. 64–78 (2014)
18. Prevosto, V.: Conception et Implantation du langage FoC pour le développement de logiciels certifiés. Ph.D. thesis, Université Paris 6, September 2003
19. Slind, K.: Another look at nested recursion. In: Aagaard, M.D., Harrison, J. (eds.) TPHOLs 2000. LNCS, vol. 1869, pp. 498–518. Springer, Heidelberg (2000)

Author Index