Towards Incremental Language Definition with Reusable Components

Damian Frölich and Thomas van Binsbergen

Informatics Institute, University of Amsterdam
{dfrolich,ltvanbinsbergen}@acm.org

July 28, 2022

lambda DAλS
28-29 JULY 2022
KRAKÓW | POLAND

Incremental programming

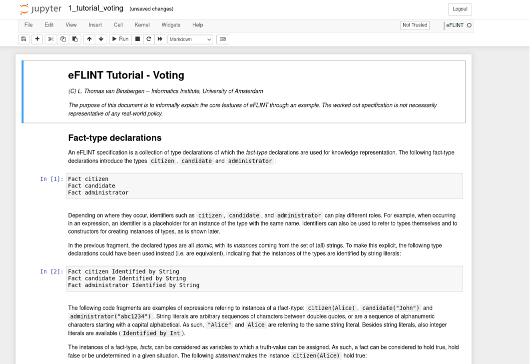
- Stepwise
- Submitting small snippets of code
- Immediate feedback

Incremental programming

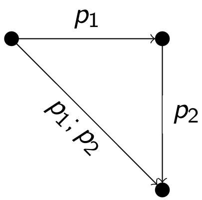
- Stepwise
- Submitting small snippets of code
- Immediate feedback
- Test new constructs (libraries, etc)
- Exploratory programming/Prototyping

Incremental programming

- Stepwise
- Submitting small snippets of code
- Immediate feedback
- Test new constructs (libraries, etc)
- Exploratory programming/Prototyping



Sequential languages[1]



[1]A principled approach to REPL interpreters. Van Binsbergen, et al.

Incremental language development

Compose languages without losing flexibility while promoting reusability

Incremental language development

Compose languages without losing flexibility while promoting reusability

- Compose language constructs
- Compose language construct semantics

Incremental language development

Compose languages without losing flexibility while promoting reusability

- Compose language constructs
- Compose language construct semantics
  $\Longrightarrow$ the Expression Problem

Incremental language development

Compose languages without losing flexibility while promoting reusability

- Compose language constructs
- Compose language construct semantics
  $\Longrightarrow$ the Expression Problem
  $\therefore$ Data types 'a la carte (DTC)

Incremental language development

Compose languages without losing flexibility while promoting reusability

- Compose language constructs
- Compose language construct semantics
  $\Longrightarrow$ the Expression Problem
  $\therefore$ Data types 'a la carte (DTC)

Incremental language development

Compose languages without losing flexibility while promoting reusability

- Compose language constructs
- Compose language construct semantics
  $\Longrightarrow$ the Expression Problem
  $\therefore$ Data types 'a la carte (DTC)
- Promote re-use and compatibility
  $\therefore$ Use funcons as a lingua franca

Incremental language development

Compose languages without losing flexibility while promoting reusability

- Compose language constructs
- Compose language construct semantics
  $\Longrightarrow$ the Expression Problem
  $\therefore$ Data types ‘a la carte (DTC)
- Promote re-use and compatibility
  $\therefore$ Use funcons as a lingua franca
- Modify the structure of the language
  $\therefore$ A new approach for language definitions

Data types 'a la carte[2]

```
data Num a = Num int
    deriving Functor
data Add a = Add a a
    deriving Functor
data Lang = Fix (Num :+: Add)
data Fix f = In (f (Fix f))
```

[2] Swierstra. JFP 2008

Data types ‘a la carte[2]

```
data Num a = Num int
    deriving Functor
data Add a = Add a a
    deriving Functor
data Lang = Fix (Num :+: Add)
data Fix f = In (f (Fix f))
```

```
foldLang :: Functor f -> (f a -> a)
      -> Fix f -> a
instance Eval Num where
    eval (Num n) = n
instance Eval Add where
    eval (Add e1 e2) = e1 + e2
```

[2] Swierstra. JFP 2008

Funcons[4]

- Component-based approach towards formal, dynamic semantics
- A library of reusable, fundamental constructs (funcons)
- Executable micro-interpreters[3]

[3] Executable Component-Based Semantics. Van Binsbergen, Sculthorpe, Mosses. JLAMP 2019
[4] PLanCompS project lead by Peter Mosses

Funcons[4]

- Component-based approach towards formal, dynamic semantics
- A library of reusable, fundamental constructs (funcons)
- Executable micro-interpreters[3]

$$\textbf{apply}(\textbf{bound}(''f''), \textbf{integer-add}(12))$$

[3] Executable Component-Based Semantics. Van Binsbergen, Sculthorpe, Mosses. JLAMP 2019
[4] PLanCompS project lead by Peter Mosses

Contribution

A method that supports incremental language definition via composition by delaying language structure choices

Language structure

$$
\begin{aligned}
Var_{\mathscr{O}} &: String \to Expr \\
Abs_{\mathscr{O}} &: String \times Expr \to Expr \\
App_{\mathscr{O}} &: Expr \times Expr \to Expr
\end{aligned}
$$

Language structure

$$\begin{aligned} Var_{\mathscr{O}} &: String \to Expr \\ Abs_{\mathscr{O}} &: String \times Expr \to Expr \\ App_{\mathscr{O}} &: Expr \times Expr \to Expr \end{aligned}$$

Language structure is decided by the types of our constructors

Delayed language structure

Operator declarations

introduces operators, arities and name 'operand positions'

$$\begin{aligned} \mathit{Var}_{\mathscr{O}} &: \mathit{VarVar} \\ \mathit{Abs}_{\mathscr{O}} &: \mathit{AbsVar} \times \mathit{AbsBody} \\ \mathit{App}_{\mathscr{O}} &: \mathit{AppAbs} \times \mathit{AppArg} \end{aligned}$$

Delayed language structure

Operator declarations

introduces operators, arities and name ‘operand positions’

$$Var_{\mathscr{O}} : VarVar$$
$$Abs_{\mathscr{O}} : AbsVar \times AbsBody$$
$$App_{\mathscr{O}} : AppAbs \times AppArg$$

Sort constraints

assign (one or more) operators to (possibly new) sorts.

Delayed language structure

Operator declarations

introduces operators, arities and name ‘operand positions’

$$Var_{\mathscr{O}} : \mathit{VarVar}$$
$$Abs_{\mathscr{O}} : \mathit{AbsVar} \times \mathit{AbsBody}$$
$$App_{\mathscr{O}} : \mathit{AppAbs} \times \mathit{AppArg}$$

Sort constraints

assign (one or more) operators to (possibly new) sorts.

Operator assignments

$$Var_{\mathscr{O}} \in \mathit{AbsBody}$$

Delayed language structure

Operator declarations

introduces operators, arities and name ‘operand positions’

$$
\begin{aligned}
\mathit{Var}_{\mathscr{O}} &: \mathit{VarVar} \\
\mathit{Abs}_{\mathscr{O}} &: \mathit{AbsVar} \times \mathit{AbsBody} \\
\mathit{App}_{\mathscr{O}} &: \mathit{AppAbs} \times \mathit{AppArg}
\end{aligned}
$$

Sort constraints

assign (one or more) operators to (possibly new) sorts.

Operator assignments

$$
\begin{aligned}
\mathit{Var}_{\mathscr{O}} &\in \mathit{AbsBody} \\
\mathit{Var}_{\mathscr{O}} &\in \mathit{Expr} \\
\mathit{App}_{\mathscr{O}} &\in \mathit{Expr} \\
\mathit{Abs}_{\mathscr{O}} &\in \mathit{Expr}
\end{aligned}
$$

Delayed language structure

Operator declarations

introduces operators, arities and name 'operand positions'

$$Var_{\mathscr{O}} : VarVar$$
$$Abs_{\mathscr{O}} : AbsVar \times AbsBody$$
$$App_{\mathscr{O}} : AppAbs \times AppArg$$

Sort constraints

assign (one or more) operators to (possibly new) sorts.

Operator assignments

$$Var_{\mathscr{O}} \in AbsBody$$
$$Var_{\mathscr{O}} \in Expr$$
$$App_{\mathscr{O}} \in Expr$$
$$Abs_{\mathscr{O}} \in Expr$$

Sub-sort constraints

$$Expr \subseteq AbsBody$$
$$Expr \subseteq AppAbs$$
$$Expr \subseteq AppArg$$

Operator semantics

$$
\begin{aligned}
\mathit{Var}_{\mathscr{F}}(\mathit{lit}) &= \textbf{bound string } \mathit{lit} \\
\mathit{Abs}_{\mathscr{F}}(x, b) &= \textbf{function closure scope}(\textbf{bind}(\textbf{string } x, \textbf{given}), b) \\
\mathit{App}_{\mathscr{F}}(\mathit{abs}, \mathit{arg}) &= \textbf{apply}(\mathit{abs}, \mathit{arg})
\end{aligned}
$$

Semantic functions translate operator occurrences to funcon terms (semantic domain).

Operator semantics

$$
\begin{aligned}
Var_{\mathscr{F}}(lit) &= \textbf{bound string } lit \\
Abs_{\mathscr{F}}(x, b) &= \textbf{function closure scope}(\textbf{bind}(\textbf{string } x, \textbf{given}), b) \\
App_{\mathscr{F}}(abs, arg) &= \textbf{apply}(abs, arg)
\end{aligned}
$$

Semantic functions translate operator occurrences to funcon terms (semantic domain).

Existing operator semantics not always enough for every language

Operator specialization

Associating 'wrapper funcon terms' as part of sort constraints

$$Return_{\mathscr{O}} : ReturnVal \qquad \text{(Operator declaration)}$$

$$Return_{\mathscr{F}}(val) = \textbf{return}\ val \qquad \text{(Semantic function)}$$

$$Return_{\mathscr{O}} \in AbsBody \qquad \text{(Sort constraint with glue code)}$$

$$\hookrightarrow \textbf{handle-return}(AbsBody_{\mathscr{F}}) \qquad \text{(glue code)}$$

Language definition

Definition

A language L is a structure $\langle O, T, S, F, G, I\rangle$ with:

O a set of operators,

$T \subset O$ the set of operators assigned to the top-level,

S a family of sets representing operator assignments

F a family of functions representing the semantic functions

G a family of glue functions

I a function for initialisation over top-level operators

Implementation

```
data Abs u t where
    Abs :: IsTrue (AbsBody t) => String -> u t -> Abs u AbsType
data AbsType
type family AbsBody t
```

Implementation

```haskell
data Abs u t where
    Abs :: IsTrue (AbsBody t) => String -> u t -> Abs u AbsType
data AbsType
type family AbsBody t
$(genSortConstraint [(''AbsType, ''AbsBody)])
```

$\Longrightarrow \mathit{Abs}_{\mathscr{O}} \in \mathit{AbsBody}$

Implementation

```
data Abs u t where
    Abs :: IsTrue (AbsBody t) => String -> u t -> Abs u AbsType
data AbsType
type family AbsBody t
$(genSortConstraint [(''AbsType, ''AbsBody)])
```

$\Longrightarrow Abs_{\mathcal{O}} \in AbsBody$

```
type family Expr t
$(genSortConstraint [(''AbsType, ''Expr)])
$(genSubSort [(''Expr, ''AbsBody)])
```

$\Longrightarrow Abs_{\mathcal{O}} \in Expr$

$Expr \subseteq AbsBody$

Semantics

```
instance ToFuncons Abs where
    toFuncons (Abs s (K body)) = K $
        function [closure [scope [bind [string s, given], body]]]
```

Glue code

```
$(genGlue (''AbsBody, ''P(Command), ''absGlueReturn))
absGlueReturn command_f = handle_return [command_f]
```

Glue code

```
$(genGlue (''AbsBody, ''P(Command), ''absGlueReturn))
absGlueReturn command_f = handle_return [command_f]
```

$\Longrightarrow$

```
instance GetGlue AbsGlue Command Funcons where
    getGlue AbsVarGlue  _ = id
    getGlue AbsBodyGlue l = absGlueReturn

class GetGlue operand (f :: (* -> *) -> * -> *) target where
    getGlue :: operand -> f (Term a) b -> target -> target
    getGlue _ _ = id
```

Why we delay

```
data Language = Language
{ operators :: [Operator]
, op_assign :: [OperatorAssignment]
, sub_sorts :: [SubSort]
, glue_code :: [(Sort, MetaType, GlueFunction)]
, init_code :: [(MetaType, GlueFunction)]
} deriving (Show)
```

Why we delay

```
data Language = Language
{ operators :: [Operator]
, op_assign :: [OperatorAssignment]
, sub_sorts :: [SubSort]
, glue_code :: [(Sort, MetaType, GlueFunction)]
, init_code :: [(MetaType, GlueFunction)]
} deriving (Show)
```

```
type family Expr
lambdaLanguage = Language
{operators =
  [(''Var, ''VarType), (''Abs, ''AbsType),
   (''App, ''AppType)]
,op_assign =
  [(op, ''Expr) | op <-
    [''VarType, ''AbsType, ''AppType]]
,sub_sorts =
  [(''Expr, t) | t <-
    [''AbsBody, ''AppLeft, ''AppRight, ''TopLevel]],
...
})
```

Why we delay

```
data Language = Language
{ operators :: [Operator]
, op_assign :: [OperatorAssignment]
, sub_sorts :: [SubSort]
, glue_code :: [(Sort, MetaType, GlueFunction)]
, init_code :: [(MetaType, GlueFunction)]
} deriving (Show)
```

```
type family Expr
lambdaLanguage = Language
{operators =
  [(''Var, ''VarType), (''Abs, ''AbsType),
   (''App, ''AppType)]
,op_assign =
  [(op, ''Expr) | op <-
    [''VarType, ''AbsType, ''AppType]]
,sub_sorts =
  [(''Expr, t) | t <-
    [''AbsBody, ''AppLeft, ''AppRight, ''TopLevel]],
...
})
```

```
<*> :: Language -> Language -> Language
```

Example

```
import Lambda(Expr)
arithLanguage = Language
{ operators = [(''Add, ''AddType), (''IntV, ''IntVType)]
, op_assign = [(op, ''Expr) | op <- [''AddType, ''IntVType]]
, sub_sorts = [(''Expr, t) | t <- [''AddLeft, ''AddRight]]
...
}
```

Can now extend the lambda language with the arithmetic language (extension)

```
lambdaLanguage <*> arithLanguage
```

Example

```
import Lambda(Expr)
arithLanguage = Language
{ operators = [(''Add, ''AddType), (''IntV, ''IntVType)]
, op_assign = [(op, ''Expr) | op <- [''AddType, ''IntVType]]
, sub_sorts = [(''Expr, t) | t <- [''AddLeft, ''AddRight]]
...
}
```

Can now extend the lambda language with the arithmetic language (extension)

```
lambdaLanguage <*> arithLanguage
```

Without a dependency (unification)

```
lambdaLanguage <*> arithLanguage <*> arithLambdaGlue
```

Example

```
import Lambda(Expr)
arithLanguage = Language
{ operators = [(''Add, ''AddType), (''IntV, ''IntVType)]
, op_assign = [(op, ''Expr) | op <- [''AddType, ''IntVType]]
, sub_sorts = [(''Expr, t) | t <- [''AddLeft, ''AddRight]]
...
}
```

Can now extend the lambda language with the arithmetic language (extension)

```
lambdaLanguage <*> arithLanguage
```

Without a dependency (unification)

```
lambdaLanguage <*> arithLanguage <*> arithLambdaGlue
```

With restriction

```
functional = lambdaLanguage <*> arithLanguage <*> modExceptionsLanguage
    where
        modExceptionsLanguage = exceptionsLanguage {
            op_assign = removeCatch $ op_assign modExceptionsLanguage
            sub_sorts = removeCatch $ sub_sorts modExceptionsLanguage }
```

Example

```
import Lambda(Expr)
arithLanguage = Language
{ operators = [(''Add, ''AddType), (''IntV, ''IntVType)]
, op_assign = [(op, ''Expr) | op <- [''AddType, ''IntVType]]
, sub_sorts = [(''Expr, t) | t <- [''AddLeft, ''AddRight]]
...
}
```

Can now extend the lambda language with the arithmetic language (extension)

```
lambdaLanguage <*> arithLanguage
```

Without a dependency (unification)

```
lambdaLanguage <*> arithLanguage <*> arithLambdaGlue
```

With restriction

```
functional = lambdaLanguage <*> arithLanguage <*> modExceptionsLanguage
    where
        modExceptionsLanguage = exceptionsLanguage {
            op_assign = removeCatch $ op_assign modExceptionsLanguage
            sub_sorts = removeCatch $ sub_sorts modExceptionsLanguage }
```

Correspond to the operations used by Erdweg

Conclusion

Approach

- \+ Highly flexible
- \+ Permits rapid prototyping
- \- Difficult to track composition effects

Implementation

- \+ Properties are statically checked by Haskell
- \+ Languages are just Haskell data types
- \+ Semantic functions are just Haskell functions
- \- Interactivity hampered by Template Haskell
- \- Language definitions can be verbose

Conclusion

Approach

+ Highly flexible
+ Permits rapid prototyping
- Difficult to track composition effects

Implementation

+ Properties are statically checked by Haskell
+ Languages are just Haskell data types
+ Semantic functions are just Haskell functions
- Interactivity hampered by Template Haskell
- Language definitions can be verbose

Future work

- Development of an external DSL to fully utilise interactivity
- User defined concrete syntax
- Explore other algebras

Towards Incremental Language Definition with Reusable Components

Damian Frölich and Thomas van Binsbergen

Informatics Institute, University of Amsterdam
{dfrolich,ltvanbinsbergen}@acm.org

July 28, 2022

lambda DAΛS
28-29 JULY 2022
KRAKÓW | POLAND