

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



Incremental programming

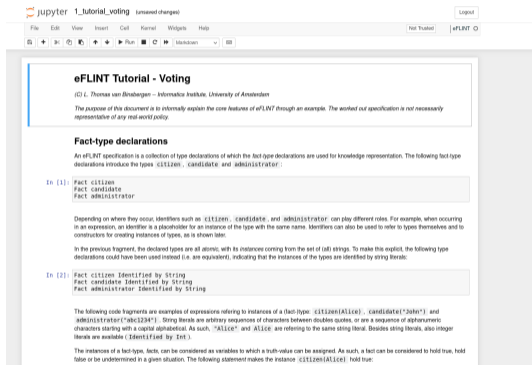
- Stepwise
- Submitting small snippets of code
- Immediate feedback

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



The screenshot shows a Jupyter Notebook interface with the following content:

eFLINT Tutorial - Voting
(C) L. Thomas van Binsbergen – Informatica Institute, University of Amsterdam

The purpose of this document is to informally explain the core features of eFLINT through an example. The worked out specification is not necessarily representative of any real-world policy.

Fact-type declarations

An eFLINT specification is a collection of type declarations of which the fact type declarations are used for knowledge representation. The following fact type declarations introduce the types `citizen`, `candidate` and `administrator`:

```
In [1]: Fact citizen
        Fact candidate
        Fact administrator
```

Depending on where they occur, identifiers such as `citizen`, `candidate`, and `administrator` can play different roles. For example, when occurring in an expression, an identifier is a placeholder for an instance of the type with the same name. Identifiers can also be used to refer to types themselves and to constructors for creating instances of types, as is shown later.

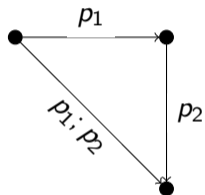
In the previous fragment, the declared types are all atomic, with its instances coming from the set of (all) strings. To make this explicit, the following type declarations could have been used instead (i.e. are equivalent), indicating that the instances of the types are identified by string literals:

```
In [2]: Fact citizen Identified by String
        Fact candidate Identified by String
        Fact administrator Identified by String
```

The following code fragments are examples of expressions referring to instances of a fact-type: `citizen(Alice)`, `candidate("John")` and `administrator("abc1234")`. String literals are arbitrary sequences of characters between double quotes, or are a sequence of alphanumeric characters starting with a capital alphabetical. As such, "Alice" and ALICE are referring to the same string literal. Besides string literals, also integer literals are available (Identified by Int).

The instances of a fact-type, facts, can be considered as variables to which a truth-value can be assigned. As such, a fact can be considered to hold true, hold false or be undetermined in a given situation. The following statement makes the instance `citizen(Alice)` hold true:

Sequential languages¹



¹A 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
⇒ the Expression Problem

Incremental language development

Compose languages without losing flexibility while promoting reusability

- Compose language constructs
- Compose language construct semantics
⇒ the Expression Problem
∴ Data types 'a la carte (DTC)

Incremental language development

Compose languages without losing flexibility while promoting reusability

- Compose language constructs
- Compose language construct semantics
 - ⇒ the Expression Problem
 - ∴ Data types 'a la carte (DTC)

Incremental language development

Compose languages without losing flexibility while promoting reusability

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

Incremental language development

Compose languages without losing flexibility while promoting reusability

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

Data types 'a la carte²

```
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))
```

²Swierstra. JFP 2008

Data types 'a la carte²

```
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
```

²Swierstra. JFP 2008

Funcons⁴

- Component-based approach towards formal, dynamic semantics
- A library of reusable, fundamental constructs (funcons)
- Executable micro-interpreters³

³Executable Component-Based Semantics. Van Binsbergen, Sculthorpe, Mosses. JLAMP 2019

⁴PLanCompS project lead by Peter Mosses

Funcons⁴

- Component-based approach towards formal, dynamic semantics
- A library of reusable, fundamental constructs (funcons)
- Executable micro-interpreters³

apply(bound(" f"), integer-add(12))

³Executable Component-Based Semantics. Van Binsbergen, Sculthorpe, Mosses. JLAMP 2019

⁴PLanCompS project lead by Peter Mosses

Contribution

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

Language structure

$Var_{\emptyset} : String \rightarrow Expr$

$Abs_{\emptyset} : String \times Expr \rightarrow Expr$

$App_{\emptyset} : Expr \times Expr \rightarrow Expr$

Language structure

$$\text{Var}_\emptyset : \text{String} \rightarrow \text{Expr}$$
$$\text{Abs}_\emptyset : \text{String} \times \text{Expr} \rightarrow \text{Expr}$$
$$\text{App}_\emptyset : \text{Expr} \times \text{Expr} \rightarrow \text{Expr}$$

Language structure is decided by the types of our constructors

Delayed language structure

Operator declarations

introduces operators, arities and name 'operand positions'

$$\text{Var}_\theta : \text{VarVar}$$
$$\text{Abs}_\theta : \text{AbsVar} \times \text{AbsBody}$$
$$\text{App}_\theta : \text{AppAbs} \times \text{AppArg}$$

Delayed language structure

Operator declarations

introduces operators, arities and name 'operand positions'

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

Sort constraints

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

Delayed language structure

Operator declarations

introduces operators, arities and name 'operand positions'

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

Sort constraints

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

Operator assignments

$$Var_{\theta} \in AbsBody$$

Delayed language structure

Operator declarations

introduces operators, arities and name 'operand positions'

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

Sort constraints

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

Operator assignments

$$Var_{\theta} \in AbsBody$$
$$Var_{\theta} \in Expr$$
$$App_{\theta} \in Expr$$
$$Abs_{\theta} \in Expr$$

Delayed language structure

Operator declarations

introduces operators, arities and name 'operand positions'

$$\text{Var}_\theta : \text{VarVar}$$

$$\text{Abs}_\theta : \text{AbsVar} \times \text{AbsBody}$$

$$\text{App}_\theta : \text{AppAbs} \times \text{AppArg}$$

Sort constraints

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

Operator assignments

$$\text{Var}_\theta \in \text{AbsBody}$$

$$\text{Var}_\theta \in \text{Expr}$$

$$\text{App}_\theta \in \text{Expr}$$

$$\text{Abs}_\theta \in \text{Expr}$$

Sub-sort constraints

$$\text{Expr} \subseteq \text{AbsBody}$$

$$\text{Expr} \subseteq \text{AppAbs}$$

$$\text{Expr} \subseteq \text{AppArg}$$

Operator semantics

$Var_{\mathcal{F}}(lit) = \text{bound string } lit$

$Abs_{\mathcal{F}}(x, b) = \text{function closure scope}(\text{bind}(\text{string } x, \text{given}), b)$

$App_{\mathcal{F}}(abs, arg) = \text{apply}(abs, arg)$

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

Operator semantics

$Var_{\mathcal{F}}(lit) = \text{bound string } lit$

$Abs_{\mathcal{F}}(x, b) = \text{function closure scope}(\text{bind}(\text{string } x, \text{given}), b)$

$App_{\mathcal{F}}(abs, arg) = \text{apply}(abs, arg)$

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

Existing operator semantics not always enough for every language

Operator specialization

Associating 'wrapper funcon terms' as part of sort constraints

$Return_{\emptyset} : ReturnVal$	(Operator declaration)
$Return_{\mathcal{F}}(val) = \mathbf{return} \ val$	(Semantic function)
$Return_{\emptyset} \in AbsBody$	(Sort constraint with glue code)
$\hookrightarrow \mathbf{handle-return}(AbsBody_{\mathcal{F}})$	(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

```
data Abs u t where
  Abs :: IsTrue (AbsBody t) => String -> u t -> Abs u AbsType
data AbsType
type family AbsBody t
$(genSortConstraint [('AbsType', 'AbsBody')])
 $\implies Abs_{\emptyset} \in AbsBody$ 
```

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']))
 $\implies Abs_{\emptyset} \in AbsBody$ 
type family Expr t
$(genSortConstraint [('AbsType', 'Expr']))
$(genSubSort [('Expr', 'AbsBody']))
 $\implies Abs_{\emptyset} \in Expr$ 
 $Expr \subseteq AbsBody$ 
```

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]
```

⇒

```
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)
```

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

```
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]],
...
})
```

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

