The Theory Plug-in: a tutorial

Presented by Andy Edmunds

Issam Maamria

University of Southampton

February 5, 2013

Outline

Motivation

1. Provide a mechanism to extend the mathematical language.
2. Provide a mechanism to extend the proof infrastructure.
3. Ensure that both mechanisms do not compromise soundness of the formalism.
4. Relieve the end-user from having to write code for simple extensions.
5. Ensure practicality and ease of use in any resultant tools.

The Theory Plug-in $\rightarrow$ Mathematical Extensions

A theory can define:

- Datatypes,
- Operators,
- Polymorphic theorems,
- Rewrite and inference rules.

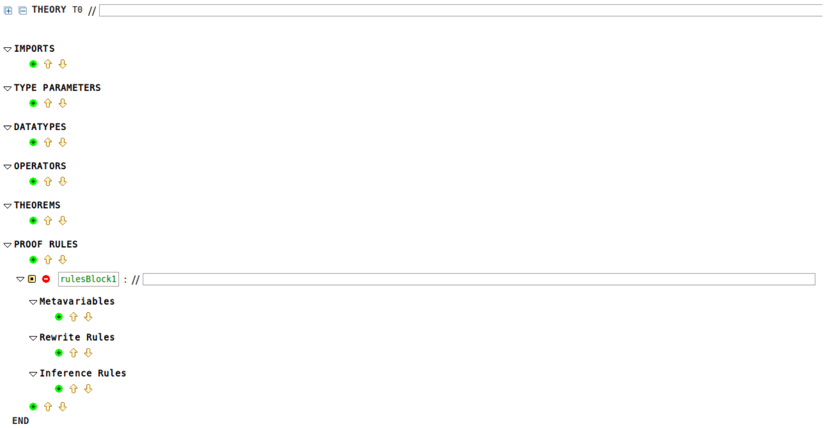
The Theory Plug-in → Capabilities

- A theory component is the place holder for defining new ‘extensions’.
- Theories are polymorphic on the type parameters which they define.
- Theories are subject to static checking and proof obligation generation.
- Proof obligations generated from theories aim to ensure soundness is not compromised.

The Theory Plug-in → Capabilities

- Theory *Deployment* takes theories from ‘development mode’ to ‘production mode’.
- At this stage users should have discharged all proof obligations that have been generated by the theory plug-in.
- IMPORT establishes a partial order between theories.
- Once a theory is deployed, no further work is required from the user. Extensions are immediately usable in models and proofs.
- The plug-in does all the work related to: polymorphism/pattern matching, type checking, well-definedness ...

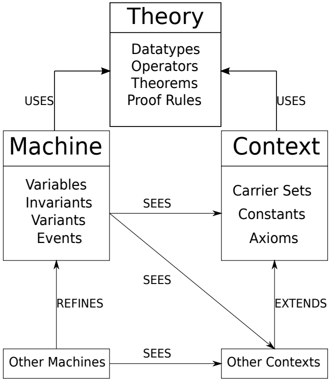
The Theory Component



- We will explain each sub-element of the theory.

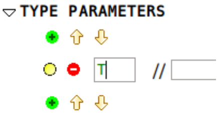
The Theory Component

- Machines and Contexts *Use* Theory Components
  - But, there is no explicit ‘uses’ clause.



The Theory Component

- Theories are polymorphic on the type parameters which they define.
- A type parameter is a set just like a carrier set in a context.
- The only assumption is that it is not empty.
- A type parameter can be instantiated with any type.
- Potential type instantiation for a type parameter T:
  - $\mathbb{Z}$
  - $\mathbb{P}(\mathbb{Z} \times BOOL)$
  - $TEMP \times (\mathbb{Z} \leftrightarrow JOB)$ where $TEMP$ and JOB are carrier sets.



A new Event-B polymorphic operator can be defined by providing the following information in the Theory UI:

1. *Parser Information*: this includes the syntax, the notation (infix or prefix), and the syntactic class (term or formula).
2. *Type Checker Information*: this includes the types of the child arguments, and the resultant type if the operator is a term operator.
3. *Prover Information*: this includes the well-definedness of the operator as well as its definition which may be used to reason about it.

Parser information includes:

1. The syntax symbol, which must be distinct from any previously recognised symbol.
2. The notation: We currently support infix and prefix.
3. The syntactic class: such as,
   - an expression operator (like *card*) or,
   - a predicate operator (like *finite*).

Type checker information includes:

1. The type of any child arguments of the operator.
   - The type of the only child argument of *card* is a power set of any type $\mathbb{P}(T)$.
2. The resultant type of the operator **if** it is an expression operator.
   - The resultant type of *card* is $\mathbb{Z}$.

$$\frac{\mathbf{type}(s) \ = \ \mathbb{P}(T)}{\mathbf{type}(card(s)) \ = \ \mathbb{Z}} \ type_{card}$$

The Theory Component $\rightarrow$ Adding New Operators

Prover information includes:

1. The definition of the operator.
   1. Currently, only direct and recursive definitions are supported.
   2. The definition may only refer to the arguments of the operator and their types.
   3. The definition may be used in proofs as a rewrite.
2. The well-definedness condition to be used.
   1. This will be used (and instantiated) when generating proof obligations.
   2. If no condition is supplied by the user, the well-definedness condition is generated from the direct definition.
3. Associativity/commutativity properties.

Proof obligations are generated to show:

- Well-definedness; ensuring that the user-supplied well-definedness condition is stronger than the well-definedness condition of the operator's direct definition.
- Commutativity; ensuring that the operator is commutative.
- Associativity; ensuring that the operator is associative.

Associativity and commutativity properties are also used to facilitate pattern matching.

---

operator *name*

  (**prefix** | **infix**) (**expression** | **predicate**)

  args $x_1 \in T_{x_1}, ..., x_n \in T_{x_n}$

  condition $P(x_1, ..., x_n)$

  definition $Q(x_1, ..., x_n)$

An Example:

---

theory *SeqThy*

type parameters T

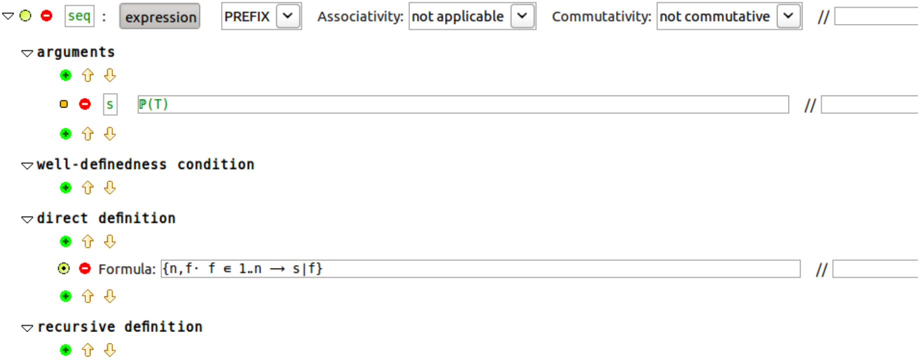
operator *Seq* **expression prefix**

  args $a \in \mathbb{P}(T)$

  condition $\top$

  definition $\{f, n \cdot f \in 1..n \to a \mid f\}$

The Theory Component → A New Sequence



- Conceptually, polymorphic theorems are Event-B predicates that have no free variables.
- They are polymorphic on the type parameters that occur in them.
- Sometimes, theorems can be monomorphic if they only refer to the existing types $\mathbb{Z}$ and $BOOL$.
- Examples:
  1. $\forall a : \mathbb{P}(A), b : \mathbb{P}(A) \cdot a \subseteq b \Rightarrow (finite(b) \Rightarrow finite(a))$,
  2. $\forall f : A \leftrightarrow B, a : \mathbb{P}(A), b : \mathbb{P}(B) \cdot f \in a \nrightarrow b \Rightarrow (finite(a) \Rightarrow finite(f))$,
  3. $\forall x, y . x * y = 0 \Rightarrow (x = 0 \vee y = 0)$.

The Theory Component → Polymorphic Theorems

- Proof obligations generated for theorems:
  1. Well-definedness: ensures that the theorem is well-defined.
  2. Validity: ensures that the theorem is valid.
- Similar to Theorems in a Context.
- Theorems can be used in proofs by:
  - selecting the appropriate theorem in the UI.
  - providing an appropriate type instantiation, which refers only to types that are recognised by the current sequent.
- Once a theorem is appropriately instantiated, it will be added as a selected hypothesis in the current sequent.

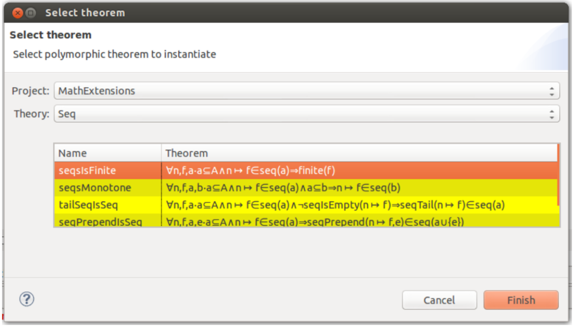
An important use of theorems is the validation of newly added operators. In the case of a sequence operator, we may add the following theorems to ensure the definition captures our understanding:

$$// \; \textit{Seq(a) may be empty}$$
$$\forall a \in \mathbb{P}(S) \cdot \varnothing \in Seq(a) \; ,$$

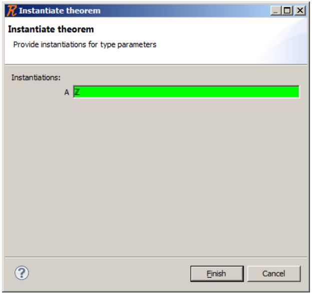
$$// \; \textit{array of finite elements}$$
$$\forall a \in \mathbb{P}(S) \cdot (\forall s \in \mathbb{P}(\mathbb{Z} \times S) \cdot s \in Seq(a) \Rightarrow finite(s)) \; .$$

Needless to say that both these theorems can also be instantiated and used in proofs.

The Theory Component → Instantiate in a Proof



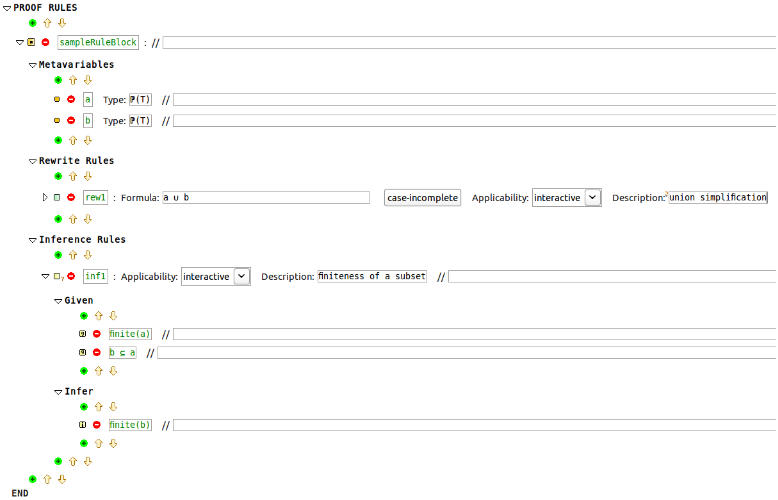
(a) Select a theorem



(b) Instantiate a theorem

The Theory Component → Proof Rules

- The theory components allows the definition of two types of proof rules:
  1. *Rewrite rules:* equalities or equivalences that can be used to rewrite predicates and expressions in sequents.
  2. *Inference rules:* which can be used to discharge, split or add more hypotheses to sequents.
- Proof rules may refer to meta-variables, and type parameters.
- Meta-variables are used to facilitate type inference/checking. Each meta-variable has a type.
- The rules clause may contain meta-variables, rewrite and inference rules. A theory may contain a number of blocks.


PROOF RULES
sampleRuleBlock : //
Metavariables
a Type: ℙ(T) //
b Type: ℙ(T) //
Rewrite Rules
rew1 : Formula: a ∪ b
case-incomplete
Applicability: interactive
Description: union simplification
Inference Rules
inf1 : Applicability: interactive
Description: finiteness of a subset //
Given
finite(a) //
b ⊆ a //
Infer
finite(b) //
END

Rewrite Rules:

- are based on equalities and equivalences.
- are polymorphic. But, the plug-in handles instantiation.
- can be applied to the goal or hypotheses.
- Examples:
  1. $E \in \{F\} \ \widehat{=} \ E = F$,
  2. $union(\mathbb{P}(S)) \ \widehat{=} \ S$,
  3. $dom(r^{-1}) \ \widehat{=} \ ran(r)$.
- Rewrite rules can be applied automatically or interactively.
  - The user decides.

rewrite *name*

[automatic] [interactive] [case complete]

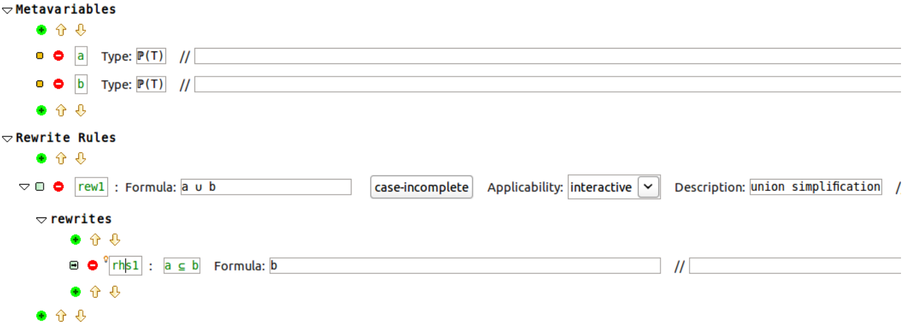
vars $x_1 : T_{x_1}, ..., x_n : T_{x_n}$

lhs $lhs(x_1, ..., x_n)$

rhs

$C_1(x_1, ..., x_n)$	$rhs_1(x_1, ..., x_n)$
...	...
$C_m(x_1, ..., x_n)$	$rhs_m(x_1, ..., x_n)$

The Theory Component → Proof Rules → Rewrite Rules



- Rewrite rules are generated from operator definitions, e.g.,

$$seq(s) \;\widehat{=}\; \{f, n \cdot f \in 1..n \rightarrow a \mid f\} \;.$$

- Proof obligations generated for rewrite rules:
  1. *Well-definedness preservation:* ensures that well-definedness is not lost when rewriting the left hand side by the right hand side of the rule.
  2. *Equality/Equivalence:* ensures that the rules sides are equal/equivalent under the stipulated conditions.

Inference Rules:

- can be used to discharge, split or add hypotheses to a sequent.
- are polymorphic. But, the plug-in handles instantiation.
- can be applied automatically or interactively.
- can be applied in a backward as well as forward fashion.
- as defined in the theory component are a convenient way of applying universally quantified implicative polymorphic theorems.

inference *name*

[automatic] [interactive]

vars $x_1 : T_{x_1}, ..., x_n : T_{x_n}$

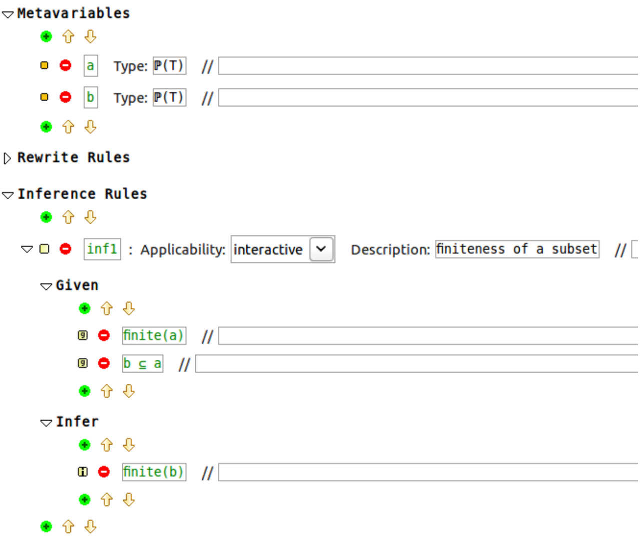
given $H_1, ..., H_m$

infer I

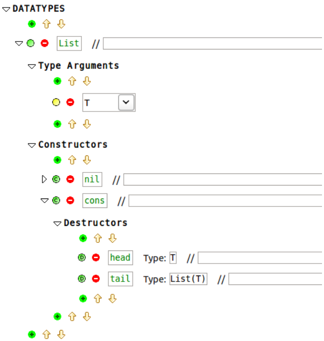
- The previous inference rule can be read in two ways:
  1. Forward Inference: if you have hypotheses H_1,..., H_m, you also have hypothesis I.
  2. Backward Inference: if you want to prove I, it is sufficient to prove each of H_1,..., H_m.
- The above inference rule can be viewed as a polymorphic theorem:

$$\forall \vec{x} \cdot \bigwedge_{i=1}^{m} H_i \ \Rightarrow \ I \tag{1}$$

- An inference rule is considered sound if its polymorphic theorem (1) is well-defined and valid.


Metavariables
a Type: ℙ(T) //
b Type: ℙ(T) //
Rewrite Rules
Inference Rules
infl : Applicability: interactive Description: finiteness of a subset //
Given
finite(a) //
b ⊆ a //
Infer
finite(b) //

- The theory plug-in supports introduction of new datatypes, and recursive functions.
- A datatype definition includes a type constructor, element constructors and destructors.
- Examples:



- As a result of the above definition, the following expression are legal Event-B expressions:

1. $List(T)$
2. nil
3. $cons(x, l0)$
4. $head(l)$
5. $tail(l)$

Simple recursive functions can also be defined:

The Theory Component → Theory Deployment

Deploying the theory:

- is the activity of moving theories from ‘development stage’ to ‘production stage’.

In the development stage:

- the user develops a hierarchy of theories using the IMPORT directive.
- Theories are staticly checked; any proof obligations are generated.
- Discharging proof obligations is mandatory before deployment.

After deployment, mathematical and proof extensions are accessible in models and proofs.

The following tactics are added to the proof interface:

1. **XD**: (eXpand Definitions) this tactic allows (whenever possible) the rewrite of theory-defined operators occurring in a sequent using their definition.
2. **TH**: (polymorphic THeorem) this tactic allows the selection and instantiation of polymorphic theorems.

Conclusion

We have discussed the main features of mathematical extensions using the theory plug-in, including:

- adding new mathematical operators,
- rewrite rules,
- inference rules.
- Polymorphic theorems and their instantation.

Hands-on Session

1. Develop a theory of sequences.