A Formal Template Language enabling Metaproof

Nuno Amálio, Susan Stepney, and Fiona Polack

Department of Computer Science, University of York, York, YO10 5DD, UK
{namalio,susan,fiona}@cs.york.ac.uk

Abstract. Design patterns are usually described in terms of instances. Templates describe sentences of some language with a particular form, generate sentences upon instantiation, and can be used to describe those commonly occurring structures that make a pattern. This paper presents FTL, a language to express templates, and an approach to proof with templates. This enables reuse at the level of formal modelling and verification: patterns of models are captured once and their structure is explored for proof, so that patterns instances can be generated mechanically and proved results related with the pattern can be reused in any context. The paper uses templates to capture the Z promotion pattern and metaproof to prove properties of Z promotion. The proved properties are applicable directly to Z promotions built by template instantiation.
Keywords: patterns, templates, proof, Z, formal development.

1 Introduction

Design patterns [1] have had an impact in software engineering. There is also a growing interest in patterns for formal development (e.g. [2–5]). Patterns, however, are usually described in terms of instances, making their mechanical adaption to a context impossible. We faced this problem while using and building patterns for Z [4, 6–8], and so we resorted to *templates* to describe structural patterns. Templates capture the form (or shape) of sentences of some language, generate, upon instantiation, sentences whose form is as prescribed by the template, and can be used to describe those commonly occurring structures that make a pattern. This paper presents the formal template language (FTL), a language to express templates, a calculus for the instantiation of FTL templates, and an approach to proof with FTL templates of formal models.

Templates of some form appear often in the computer science literature. For example, a popular Z book [9, p. 150] introduces the Z schema with a template:

$$Name == [\ declaration \mid predicates\ ]$$

This is an *informal template*, it says that a schema has a name, a set of declarations and a set of predicates. Intuitively, we guess that the names of the template are to be substituted by values, which is confirmed by a template instance:

$$Bank == [accs : \mathbb{P}\ ACCID;\ accSt : ACCID \pfun Account \mid \operatorname{dom} accSt = accs]$$

The problem with informal templates is that it is difficult to distinguish the template from the instance, and is difficult to know what instantiations are valid, making it impossible to reason rigorously with them. FTL can represent templates of any formal language precisely; it is used here with Z. The informal template given above in FTL is:

$$\lessdot \mathit{Name} \gtrdot == [\ \llbracket\ \lessdot \mathit{declaration} \gtrdot\ \rrbracket \mid \llbracket\ \lessdot \mathit{predicate} \gtrdot\ \rrbracket\ ]$$

A variable within $\lessdot\gtrdot$ denotes a *placeholder*, which is substituted by a value when the template is instantiated; a term within $\llbracket\ \ \rrbracket$ denotes a list, which is replaced by many occurrences of the term in the instantiation.

1.1 Metaproof

The form of Z sentences can be represented as FTL templates. It is also possible to explore templates of Z for reasoning (or proof). This has practical value: template developers can establish *metatheorems* for templates that are applicable to all instantiations of the templates involved. This is motivated with an example.

In Z, the introduction of a state space definition of an abstract data type (ADT), such as *Bank* above, into a specification, entails a demonstration that the description is consistent: at least one state satisfying the description should exist. This normally involves defining the initial state of the ADT (the initialisation) and proving that the initial state does exist (the initialisation theorem). The *Bank* is initialised assuming that in the initial state there are no accounts:

$$\mathit{BankInit} == [\ \mathit{Bank}\ ' \mid \mathit{accs}' = \varnothing \land \mathit{accSt}' = \varnothing\ ]$$

The consistency of *Bank* is demonstrated by proving the Z conjecture, $\vdash ?\ \exists\, \mathit{BankInit} \bullet \mathrm{true}$. A proof-sketch of this conjecture is (see appendix A for the Z inference rules used):

$$\begin{aligned}
&\vdash \exists\ \mathit{BankInit} \bullet \mathrm{true}\\
&\equiv [\text{By } \exists\ \mathit{Sc} \text{ (twice)}]\\
&\vdash \exists\ \mathit{accs}' : \mathbb{P}\, \mathit{ACCID};\ \mathit{accSt}' : \mathit{ACCID} \rightarrowtail\!\!\!\!\!\!\!+\ \mathit{Account} \bullet \mathit{accs}' = \varnothing \land \mathit{accSt}' = \varnothing\\
&\equiv [\text{By one-point}]\\
&\vdash \mathrm{dom}\, \varnothing = \varnothing \land \varnothing \in \mathbb{P}\ \ \mathit{ACCID} \land \varnothing \in \mathit{ACCID} \rightarrowtail\!\!\!\!\!\!\!+\ \mathit{Account}\\
&\equiv [\text{By set theory and propositional calculus}]\\
&\mathrm{true}
\end{aligned}$$

This is proved automatically in the Z/Eves [10] theorem prover.

The *Bank* schema is an instance of a common structure of Z specifications: the state of a *promoted* ADT [9, 4]. The theorem proved above applies to the *Bank* ADT, but does it apply also to all promoted ADTs that are similar in form to *Bank*? If it does, can this result be proved once and for all?

Bank was generated from a template, but that template is too general and not useful for the kind of investigation that we want to do. Instead, we use a more restricted template representing a promoted Z ADT, of which *Bank* is an instance. A promoted ADT comprises a set of identifiers, a function mapping identifiers to state, a predicate restricting the mapping function, and a predicate representing an optional state invariant:

$$\begin{array}{l} \ll P \gg == [\ \ll ids \gg : \mathbb{P}\ \ll ID \gg ;\ \ll st \gg : \ll ID \gg \mapsto \ll S \gg \mid \\ \qquad\qquad \mathrm{dom}\ \ll st \gg = \ll ids \gg \wedge \ll I \gg\ ] \end{array}$$

(Promoted ADT's without the optional invariant, such as *Bank* above, are instantiated from this template by instantiating $\ll I \gg$ with the value *true*.)

Likewise, we represent as templates the empty initialisation of a promoted ADT, and the initialisation conjecture:

$$\begin{array}{l} \ll P \gg Init == [\ \ll P \gg\ ' \mid \ll ids \gg' = \varnothing \wedge \ll st \gg' = \varnothing\ ] \\ \vdash ?\quad \exists\ \ll P \gg Init \bullet \mathrm{true} \end{array}$$

We can reason with these templates by analysing their *welformed* instantiations. In those cases, P, id and st hold names, ID and S are sets, and I is a predicate. By expanding the template schemas using the laws of the schema calculus, and apply the one-point rule (see proof above), we get the formula,

$$\begin{array}{l} \vdash \mathrm{dom}\ \varnothing = \varnothing \wedge \varnothing \in \mathbb{P} \ll ID \gg \wedge \varnothing \in \ll ID \gg \mapsto \ll S \gg \\ \qquad \wedge \ll I \gg' [\ll ids \gg' := \varnothing, \ll st \gg' := \varnothing] \end{array}$$

which reduces to, $\ll I \gg' [\ll ids \gg' := \varnothing, \ll st \gg' := \varnothing]$. If $\ll I \gg$ is instantiated with *true*, then the formula reduces to *true*. This establishes two metatheorems, where the latter is a specialisation (or a corollary) of the former, that are applicable to all promoted ADTs instantiated from these templates. The specialised metatheorem gives the nice property of *true by construction*: whenever these templates are instantiated, such that $\ll I \gg$ is instantiated with *true*, then the initialisation conjecture is simply *true*. Even when $\ll I \gg$ is not instantiated with *true*, the formula to prove is simpler than the initial one.

The argument outlined above is rigorous and valid, but it is not formal. To work towards formal metaproof, so that tool support is possible, a formal semantics for the template language is required.

In the following, FTL is given a brief introduction followed by an overview of its formal definition. Then, the instantiation calculus of FTL, a calculus for the partial instantiation of templates, is presented. Finally, a metaproof approach for Z based on FTL is developed and illustrated for the rigorous proof above.

2 A short introduction to FTL

FTL expresses templates that can be instantiated to yield sentences of some language (the target language). FTL is general in the sense that it can capture the form (or shape) of sentences of any formal language, and although designed with Z in mind, it is not tied to Z or to any other language.

FTL's abstraction mechanism is based on *variables*, which allow the representation of variation points in structures. Variables have, in FTL, their usual mathematical meaning: they denote some value in a scope. Template instantiation is, essentially, substitution of variables by values.

As FTL is general, it is possible to instantiate a template so that the resulting sentence is meaningless. FTL templates assist the author; they do not remove the obligation to check that what the author writes is sensible.

The following illustrates the main constructs of FTL.

Text. A template may contain text of the target language, which is present in every instance. For example, the trivial template, *true*, always yields, *true*, when instantiated. Usually, templates comprise text combined with other constructs.

Placeholder. A placeholder is represented by enclosing one variable within $\lll \ggg$. Placeholders, when not within lists, denote one variable occurrence, and they are substituted by the value assigned to the variable when the template is instantiated. The template, $\lll x \ggg : \lll t \ggg; \ \lll y \ggg : \lll t \ggg$, includes four placeholders and three variables, x, t and y; this can be instantiated with the substitution set, $\{x \mapsto \text{“}a\text{”}, t \mapsto \text{“}\mathbb{N}\text{”}, y \mapsto \text{“}b\text{”}\}$, to yield: $a : \mathbb{N}; \ b : \mathbb{N}$.

List. A list comprises one list term, a list separator (the separator of the instantiated list terms) and a string representing the empty instantiation of the list, and it is represented by enclosing the list term within $[\![\ \ ]\!]$. The list term is a combination of text, parameters and possibly other lists. Often, the abbreviated form of lists, without separator and empty instantiation, is used.

A placeholder within a list denotes an *indexed set* of variable occurrences. This means that $\lll x \ggg$ and $[\![\lll x \ggg]\!]$ actually denote different variable occurrences; $\lll x \ggg$ denotes an occurrence of the variable x, but $[\![\lll x \ggg]\!]$ denotes the occurrence of the indexed set of variables, $\{x_1, \ldots, x_n\}$.

The template, $[\![\lll x \ggg : \lll t \ggg]\!]_{(\text{“}; \text{”}, \text{“}\{\}\text{”})}$, can be instantiated with the sequence of substitution sets, $\langle \{x \mapsto \text{“}a\text{”}, t \mapsto \text{“}\mathbb{N}\text{”}\}, \{x \mapsto \text{“}b\text{”}, t \mapsto \text{“}\mathbb{P}\ \mathbb{N}\text{”}\} \rangle$. This yields: $a : \mathbb{N}; \ b : \mathbb{P}\,\mathbb{N}$. Lists can be instantiated with an empty instantiation, $\langle \ \rangle$; here this gives $\{\}$, the list's empty instantiation.

Choice. The FTL choice construct expresses *choice* of template expressions. That is, only one of the choices is present in the instantiation. There are two kinds of choice: optional and multiple. In optional, the single expression may be present in the instantiation or not. In multiple, one of the choices must be present in the instantiation. Choices are instantiated with a choice-selection, a natural number, indicating the selected choice; non-selection takes the value zero.

The template $\llparenthesis \lll x \ggg : \lll t \ggg \rrparenthesis^?$ can be instantiated with $(1, \{x \mapsto \text{“}a\text{”}, t \mapsto \text{“}\mathbb{N}\text{”}\})$, to yield: $a : \mathbb{N}$. To avoid the presence of the expression in the instantiation, the template can be instantiated with $(0, \{\})$, which simply yields the empty string. In the multiple-choice template,

$$\llparenthesis \lll x \ggg : \lll t \ggg \ [\!] \ \lll x \ggg : \lll t_1 \ggg \mathrel{+\!\!\!\rightarrow} \lll t_2 \ggg \rrparenthesis$$

the first choice is instantiated with $(1, \{x \mapsto \text{“}a\text{”}, t \mapsto \text{“}\mathbb{N}\text{”}\})$ to yield: $a : \mathbb{N}$; the second with $(2, \{x \mapsto \text{“}f\text{”}, t_1 \mapsto \text{“}\mathbb{N}\text{”}, t_2 \mapsto \text{“}\mathbb{N}\text{”}\})$ to yield: $f : \mathbb{N} \nrightarrow \mathbb{N}$.

3 The Formal Definition of FTL

FTL has a denotational semantics [11] based on its abstract syntax. The instantiation of an FTL template should give sentences in some target language, and this is naturally represented in the domain of strings (the semantic domain).

FTL is fully specified in Z elsewhere [12]. Here, we present its definitions in a form that is easier to read, which combines Z sets and operators, the BNF (Backus-Naur form) notation and an equational style.

3.1 Syntax

This section defines the syntactic sets of the language. The set of all identifiers (I) gives variables to construct placeholders. The set of all text symbols ($SYMB$) is used to construct the set of all strings, which are sequences of text symbols:

$$[I, SYMB] \qquad\qquad Str == \operatorname{seq} SYMB$$

The remaining syntactic sets are defined by structural induction, using the BNF notation. The set of template expressions comprises objects that are either an atom (A), a choice (C), or either of these followed by another expression:

$$E ::= A \mid C \mid A\ E \mid C\ E$$

The set of choices comprises optional and multiple choice; optional is formed by one expression, and multiple by a sequence of expressions (set CL):

$$C ::= \llparenthesis\ E\ \rrparenthesis^{?} \mid \llparenthesis\ CL\ \rrparenthesis \qquad\qquad CL\ ::=\ E_1\ \talloblong\ E_2 \mid E\ \talloblong\ CL$$

The set of atoms, A, comprises placeholders, text (T), and lists (L); a placeholder is formed by an identifier, the name of a variable:

$$A ::= \lessdot I \gtrdot \mid T \mid L$$

The set of lists, L, comprises two forms of list: normal and abbreviated. A normal list comprises a list term (set LT, a sequence of atoms), a list separator (SEP) and the empty instantiation of the list (EI); the abbreviated form just includes the list term:

$$L\ ::= [\![\ LT\ ]\!]_{(SEP,EI)} \mid [\![\ LT\ ]\!] \qquad\qquad LT\ ::= A \mid A\ LT$$

List separators, list empty instantiations and text are just strings:

$$SEP ::= Str \qquad\qquad EI ::= Str \qquad\qquad T ::= Str$$

3.2 Semantics

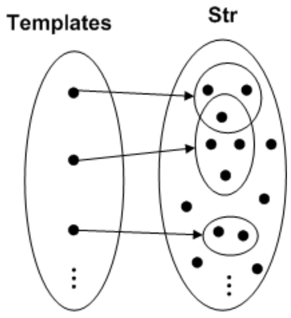

Fig. 1. Templates and the sets of strings they denote (all possible instances).

A template denotes a set of strings, corresponding to all its possible instances (Fig. 1).[1] The semantics of FTL could be specified by calculating all possible instances of a template. That is, given the set of template expressions (E) defined above, the meaning of a template would be given by the function:

$$\mathcal{M} : E \rightarrow \mathbb{P}\ Str$$

But this does not explain *instantiation*, that is, how users generate sentences from templates.

The same meaning of templates as denoting a set of strings, can be achieved by considering instantiation, which consists of substitutions for the template's variables and selections for the template's choices. A template has a set of all possible instantiations; by instantiating a template with these instantiations we get the set of all strings denoted by the template.

So, instead, the semantics is defined by an instantiation function, which calculates the string (sentence) generated by instantiating a template with an instantiation. That is:

$$\mathcal{M} : E \rightarrow TInst \nrightarrow Str$$

The semantic functions are defined by structural induction on the syntax of FTL. The following presents the semantics for atoms and expressions; the complete definitions are given in [12].

Semantics of atoms. Atoms are instantiated with substitutions for the variables that occur within placeholders, which may stand-alone or be within lists.

The environment structure (Env), defined as a partial function from identifiers (variables) to strings (values), represents a set of variable substitutions:

$$Env == I \nrightarrow Str$$

This allows the instantiation of placeholders that are not within lists. For example, the template $\lessdot x \gtrdot : \lessdot t \gtrdot$ is instantiated with, $\{x \mapsto a, t \mapsto \mathbb{N}\}$, an instance of Env, to yield $x : \mathbb{N}$. As a placeholder within a list denotes an indexed set of variable occurrences, it seems natural to instantiate these variables with a sequence of substitution sets (seq Env), but this does not work with nested lists. So, a recursive structure is required, the environment tree:

$$TreeEnv ::= \mathsf{tree}\ \langle\langle Env \times \mathrm{seq}\ TreeEnv \rangle\rangle$$

[1] But only a subset of these strings has a meaning in the target language.

TreeEnv comprises one environment, to instantiate the placeholders that stand alone in the current scope, and a sequence of *TreeEnv* to instantiate the placeholders that are within the lists of the current scope. See [12] for further details.

The semantic function extracts the required substitutions from this structure; it takes an atom (A) and a *TreeEnv*, and returns a string (*Str*):

$$\mathcal{M}_{\mathcal{A}} : A \rightarrow \mathit{TreeEnv} \nrightarrow \mathit{Str}$$

This is defined by the equations (where $e \in \mathit{Env}$ and $\mathit{ste} \in \operatorname{seq} \mathit{TreeEnv}$):

$$\begin{aligned}
&\mathcal{M}_{\mathcal{A}}(\ T\ )(\mathsf{tree}(e, \mathit{ste})) = T \\
&\mathcal{M}_{\mathcal{A}}(\ \lessdot I \gtrdot\ )(\mathsf{tree}(e, \mathit{ste})) = \begin{cases} e(I) & \text{if } I \in \operatorname{dom} e \\ \mathit{undefined} & \text{otherwise} \end{cases} \\
&\mathcal{M}_{\mathcal{A}}(\ L\ )(\mathsf{tree}(e, \mathit{ste})) = \mathcal{M}_{\mathcal{L}}(\ L\ )\ \mathit{ste}
\end{aligned}$$

If the atom is a piece of text (a string), then the text is returned. If the atom is a placeholder, then either there is a substitution for the placeholder's variable in the current environment (e) and the substitution is returned, or otherwise and the function is undefined. If the atom is a list, then the list instantiation function is called in the current sequence of environment trees (*ste*).

Semantics of expressions. The global environment (*GEnv*) structure, which represents a total template instantiation, builds on the *TreeEnv* structure; ; it comprises a sequence of natural numbers, the selections of the template's choices, and a *TreeEnv* structure, the substitutions of the template's variables:

$$\mathit{GEnv} == \operatorname{seq} \mathbb{N} \times \mathit{TreeEnv}$$

The semantic function for template expressions takes an expression (E) and a *GEnv* and returns a string:

$$\mathcal{M}_{\mathcal{E}} : E \rightarrow \mathit{GEnv} \nrightarrow \mathit{Str}$$

The equation definitions for expressions made up of atoms (see [12] for choice) is (where $\mathit{chs} \in \operatorname{seq} \mathbb{N}$ and $\mathit{te} \in \mathit{TreeEnv}$):[2]

$$\begin{aligned}
&\mathcal{M}_{\mathcal{E}}(\ A\ )(\mathit{chs}, \mathit{te}) = \mathcal{M}_{\mathcal{A}}(\ A\ )\ \mathit{te} \\
&\mathcal{M}_{\mathcal{E}}(\ A\ E\ )(\mathit{chs}, \mathit{te}) = \mathcal{M}_{\mathcal{A}}(\ A\ )\ \mathit{te} +\!\!+ \mathcal{M}_{\mathcal{E}}(\ E\ )(\mathit{chs}, \mathit{te})
\end{aligned}$$

If the expression is an atom, then the atom is instantiated in the environment tree (call to $\mathcal{M}_{\mathcal{A}}$ with *te*). If the expression is an atom followed by another expression, then the instantiation of the atom (call to $\mathcal{M}_{\mathcal{A}}$ with *te*) is concatenated with the instantiation of the rest of the expression (recursive call to $\mathcal{M}_{\mathcal{E}}$).

[2] The operator $+\!\!+$ denotes string concatenation (defined as sequence concatenation).

3.3 Illustration: Instantiating templates using the semantics

The language definition can be used to instantiate templates. This is illustrated here for the template of the promoted Z ADT:

$$\begin{aligned} &\mathit{TPADT} == \lessdot P \gtrdot == [\ \lessdot \mathit{ids} \gtrdot : \mathbb{P} \lessdot \mathit{ID} \gtrdot;\ \lessdot \mathit{st} \gtrdot : \lessdot \mathit{ID} \gtrdot \pfun \lessdot S \gtrdot \mid \\ &\qquad \operatorname{dom}\ \lessdot \mathit{st} \gtrdot = \lessdot \mathit{ids} \gtrdot \land \lessdot I \gtrdot] \end{aligned}$$

First, the substitution set for the template is specified in the environment e:

$$\begin{aligned} &e == \{P \mapsto \text{“}\mathit{Bank}\text{”}, \mathit{ids} \mapsto \text{“}\mathit{accs}\text{”}, \mathit{st} \mapsto \text{“}\mathit{accst}\text{”}, \mathit{ID} \mapsto \text{“}\mathit{ACCID}\text{”}, \\ &\quad S \mapsto \text{“}\mathit{Account}\text{”}, I \mapsto \text{“}\mathit{true}\text{”}\} \end{aligned}$$

The environment tree and the global environment build upon e. As there are no lists or choices, the sequences of trees and choice selections are empty:

$$t == \mathsf{tree}(e, \langle\rangle) \qquad\qquad g == (\langle\rangle, t)$$

The template is now instantiated by applying the semantic functions:

$$\begin{aligned} &\mathcal{M}_{\mathcal{E}}(\mathit{TPADT})(g) \\ &\equiv [\text{By defs of } \mathit{TPADT}, \mathcal{M}_{\mathcal{E}} \text{ and } g] \\ &\mathcal{M}_{\mathcal{A}}(\lessdot P \gtrdot)(t) +\!\!\!+\ \mathcal{M}_{\mathcal{E}}(==[\ \lessdot \mathit{ids} \gtrdot : \mathbb{P}\ \lessdot \mathit{ID} \gtrdot;\ \lessdot \mathit{st} \gtrdot : \lessdot \mathit{ID} \gtrdot \pfun \lessdot S \gtrdot \mid \\ &\quad \operatorname{dom}\ \lessdot \mathit{st} \gtrdot = \lessdot \mathit{ids} \gtrdot \land \lessdot I \gtrdot\ ])(g) \\ &\equiv [\text{by defs of } \mathcal{M}_{\mathcal{A}} \text{ and } \mathcal{M}_{\mathcal{E}}] \\ &e(P)\ +\!\!\!+\ \mathcal{M}_{\mathcal{A}}(\text{“} == [\ \text{”})(t) +\!\!\!+ \mathcal{M}_{\mathcal{E}}(\lessdot \mathit{ids} \gtrdot : \mathbb{P}\ \lessdot \mathit{ID} \gtrdot; \\ &\quad \lessdot \mathit{st} \gtrdot : \lessdot \mathit{ID} \gtrdot \pfun \lessdot S \gtrdot \mid \operatorname{dom}\ \lessdot \mathit{st} \gtrdot = \lessdot \mathit{ids} \gtrdot \land \lessdot I \gtrdot\ ])(g) \\ &\equiv [\text{by defs of } e, +\!\!\!+\ , \mathcal{M}_{\mathcal{A}}, \mathcal{M}_{\mathcal{E}}] \\ &\text{“}\mathit{Bank} == [\ \text{”} +\!\!\!+ \mathcal{M}_{\mathcal{A}}(\lessdot \mathit{ids} \gtrdot)(t) +\!\!\!+ \mathcal{M}_{\mathcal{E}}(: \mathbb{P} \lessdot \mathit{ID} \gtrdot; \\ &\quad \lessdot \mathit{st} \gtrdot : \lessdot \mathit{ID} \gtrdot \pfun \lessdot S \gtrdot \mid \operatorname{dom}\ \lessdot \mathit{st} \gtrdot = \lessdot \mathit{ids} \gtrdot \land \lessdot I \gtrdot\ ])(g) \end{aligned}$$

By applying the semantic functions in this way, we obtain:

$$\begin{aligned} &\text{“}\mathit{Bank} == [\ \mathit{accs} : \mathbb{P}\, \mathit{ACCID};\ \mathit{accSt} : \mathit{ACCID} \pfun \mathit{Account} \mid \\ &\qquad \operatorname{dom} \mathit{accSt} = \mathit{accs} \land \mathrm{true}]\text{”} \end{aligned}$$

3.4 Testing

FTL's semantics has been tested using the Z/Eves theorem prover, based on its Z definition. The proved theorems demonstrate that the semantic functions when applied to sample templates and instantiations yield the expected Z sentence. Test conjectures were chosen to give a good coverage of all FTL constructs and all possible instances of templates containing those constructs (see [12] for details).

The derivation presented above is demonstrated by proving the conjecture:

$$\begin{aligned} &\vdash ?\ \mathcal{M}_{\mathcal{E}}(\mathit{TPADT})(g) \\ &\quad = \text{“}\mathit{Bank} == [\ \mathit{accs} : \mathbb{P}\ \ \mathit{ACCID};\ \mathit{accSt} : \mathit{ACCID} \pfun \mathit{Account} \mid \\ &\qquad \operatorname{dom}\ \mathit{accSt} = \mathit{accs} \land \mathrm{true}]\text{”} \end{aligned}$$

And this conjecture is proved automatically in the Z/Eves prover.

4 The instantiation calculus

The semantics of FTL is defined by calculating the string (sentence) that is generated from a template given an instantiation. The instantiations of the semantics are *total*, that is, there must be substitutions for all the template's variables and selections for all the template's choices. Sometimes, however, we may be interested in *partially* instantiating a template, for example, substituting just one variable in a template and leaving the rest of the template unchanged.

The *instantiation calculus* (IC) of FTL is an approach to transform templates by taking instantiation decisions on a step-by-step basis. In this setting, a template has been fully instantiated when there no placeholders, choices or lists left; all instantiation decisions have been resolved.

To have a better idea of the calculus, consider a transformation of the promoted ADT template where the variable P is substituted with "*Bank*":

$$\begin{aligned} Bank == [\ &\ll ids \gg : \mathbb{P}\ \ll ID \gg;\ \ll st \gg : \ll ID \gg \mathrel{+\!\!\!\!\!\leftrightarrow} \ll S \gg \mid \\ &\mathrm{dom}\ \ll st \gg = \ll ids \gg \wedge \ll I \gg\ ] \end{aligned}$$

This is a more refined template, one in which the decision of substituting the variable P has been taken. By applying a similar sequence of transformations, the *Bank* schema would be reached.

The calculus is defined by instantiation functions, which take a template expression (E) and a partial instantiation, and return the template resulting from the partial instantiation of the given template:

$$I : E \to PInst \to E$$

Templates refined with the IC are just like any other FTL template: they denote a set of strings. As templates are refined with the calculus, the sets of strings they denote become smaller and smaller, until they cannot be refined any further and just denote a singleton set of strings (Fig. 2).

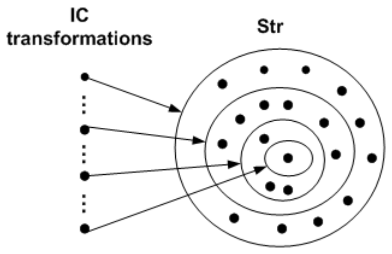

Fig. 2. Templates transformed with the instantiation calculus and the sets of strings they denote.

The IC is divided into placeholders, lists and choice. The IC functions, like in the semantics, are defined by structural induction on the syntax of FTL. The complete definitions are given elsewhere [12]. The IC was tested, in a similar fashion to the semantics (see above), by using its Z definition and the Z/Eves prover. The following presents part of the definition of the IC for placeholders.

4.1 Placeholders

The IC function simply replaces placeholders by a substitution of their variables, provided that a substitution has been provided. To represent a set of variable substitutions, the *Env* structure (see above) is reused.

The instantiation function for atoms takes an atom (A) and an environment, and returns another atom:

$$\mathcal{IP}_{\mathcal{A}} : A \rightarrow Env \rightarrow A$$

This is defined by the equations (where $e \in Env$):

$$\mathcal{IP}_{\mathcal{A}}(T)\ e = T$$

$$\mathcal{IP}_{\mathcal{A}}(\lessdot I \gtrdot)\ e = \begin{cases} e(I) & \text{If } I \in \text{dom } e \\ \lessdot I \gtrdot & \text{otherwise} \end{cases}$$

$$\mathcal{IP}_{\mathcal{A}}(L)\ e = L$$

If the atom is text then the text is returned. If the atom is a placeholder, then either there is a substitution for the variable in the environment (e) and instantiation takes place, or otherwise and the placeholder is returned uninstantiated. If the atom is a list then the list is returned; lists have their own function.

The instantiation function for expressions takes an expression (E) and an environment, and returns an expression:

$$\mathcal{IP}_{\mathcal{E}} : E \rightarrow Env \rightarrow E$$

The equations for expressions made up of atoms are:

$$\mathcal{IP}_{\mathcal{E}}(A)\ e = \mathcal{IP}_{\mathcal{A}}\ (A)\ e$$

$$\mathcal{IP}_{\mathcal{E}}(A\ E) e = (\mathcal{IP}_{\mathcal{A}}\ (A)\ e)\ \ (\mathcal{IP}_{\mathcal{E}}\ (E)\ e)$$

Essentially, the atoms that make the template expression are instantiated recursively for the given set of substitutions (e).

5 Metaproof

The formal definition of FTL and the IC can be used to support metaproof. The approach presented here is developed for Z, but it is more general; the same ideas can be applied to any language with a proof logic.

Metaproof with Z is a proof on a generalisation of a commonly occurring Z conjecture. First, the setting for metaproof with Z is presented, by discussing the link between FTL and Z for metaproof, and by considering generalisation and the concept of characteristic instantiation. Then, the approach is illustrated for the initialisation conjecture of promoted ADTs.

5.1 Linking FTL with Z

FTL is a general language: it captures the form of sentences and makes no assumptions in terms of meaning from the target language. Metaproof, however, requires FTL to be linked with the target language, so that reasoning with template representations makes sense. So, metaproof with Z considers only those templates that yield Z sentences and instances that are *welformed.* In Z, *welformed* means that the sentence is type-correct.

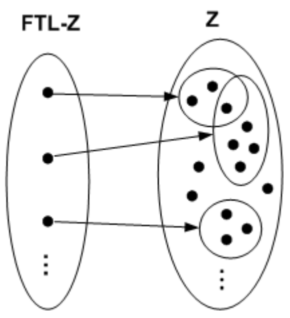

Fig. 3. FTL-Z templates denote sets of Z specifications.

Formally, given the syntactic and semantic definitions above, the set of all possible instances of a template is given by the function:

$$S == (\lambda\, t : E \bullet \{gi : GEnv \bullet \mathcal{M}_{\mathcal{E}}(t)\ gi\})$$

The function $\mathcal{ZTC}$ tells whether the given string is a type-correct Z specification:

$$\begin{array}{l} \mathcal{ZTC}\_ : \mathbb{P}\, Str \\ \mathcal{ZTC}\ s \Leftrightarrow\ s\ is\ type\ correct \end{array}$$

Then, the set of all welformed Z instances of a template, a subset of all possible instances, is given by:

$$WFS == (\lambda\, t : E \bullet \{s : S(t) \mid \mathcal{ZTC}\ s\})$$

Essentially, this is how FTL is linked with Z for metaproof: through the welformed instances of Z templates. A Z template denotes a set of Z specifications (Fig. 3), the set of all possible type-correct instances.

5.2 Characteristic instantiation and the rule of generalisation

In the process of reasoning with Z templates, there is a point where a switch from the world of templates into the world of Z occurs: a general template formula becomes an instance. This involves a *characteristic* instantiation. For example, in the template formula, $\varnothing \in \mathbb{P}{\lessdot}ID{\gtrdot}$, it is clear that ID must hold a name referring to a set; the sentence would not be type-correct otherwise. So, here, a characteristic instantiation is required: let X be an arbitrary set, instantiate ID with X to give, $\varnothing \in \mathbb{P}\, X$, which is trivially true. The set X is a characteristic instantiation of that formula. In this simple case, only one characteristic instantiation needs to be considered. Other cases require induction (for lists) or case analysis (for choice).

But, when is it safe to conclude the truth of the template statement from the proof of its instance? In formal logic there is a similar problem. Suppose a set A and a predicate P, the truth of the predicate logic statement, $\forall\ x : A \bullet P$, implies that it is true for every value in A. But how can such a statement be proved? We could prove that it is true for each value in A, but this is not practical because it may involve a large or even infinite number of proofs. This is solved by proving that P holds for an *arbitrary* member of A: if no assumptions about which member of A is chosen in order to prove P, then the proof generalises to all members. This is, in fact, a known inference rule of predicate logic called *generalisation* (or *universal introduction*) [9]:

$$\frac{\Gamma \vdash \forall\, x : A \bullet P}{\Gamma;\ x \in A\ \vdash P} \begin{array}{l}[\forall\text{-I}]\\ [x \notin FV(\Gamma)]\end{array}$$

The is the principle behind *characteristic* instantiation. An arbitrary instantiation of a template formula is introduced so that the proof of the instance generalises to the proof of the template.

5.3 A logic for template-Z

If we just use characteristic instantiation, all we can do with templates in proofs is instantiation. Only when everything has been instantiated can other inference rules be applied in proofs (see [12] for examples of this).

A better approach is to build a logic (a set of inference rules) for proof with template formulas, which extends the logic of the target language. These inference rules are proved in the logic of the target language. [12] defines a draft logic for proof with Z templates.

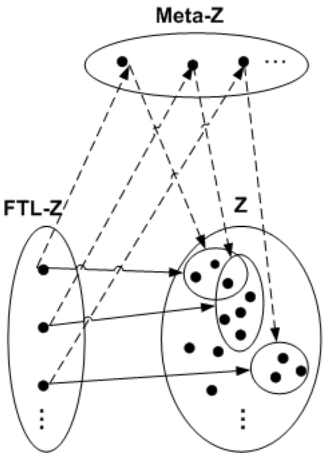

Fig. 4. A Z template denotes one meta-Z object and a set of Z specifications. A meta-Z object denotes a set of Z specifications.

In the template-Z logic, the inference rules of characteristic instantiation are, for now, axioms (unproved), which could be proved by considering a meta-world of Z (Fig. 4), where its objects represent FTL templates of Z and denote sets of Z specifications. For example, the formula, $\varnothing \in \mathbb{P}\langle\!\langle ID\rangle\!\rangle$, can be proved in meta-Z; suppose that in meta-Z there is a universal set, $\mathbb{U}$, of which all sets in a Z specification are a subset of.[3] That formula is interpreted in meta-Z as:

$$\vdash \forall ID : \mathbb{P}\,\mathbb{U} \bullet \varnothing \in \mathbb{P}\ ID$$

Now, by the law of universal introduction,

$$ID \in \mathbb{P}\,\mathbb{U} \vdash \varnothing \in \mathbb{P}\ ID$$

which is trivially true.

The following illustrates metaproof with the template-Z logic, using the initialisation conjecture of the promoted ADT.

5.4 Proof with the template-Z logic

We now go back to the proof of the initialisation of the promoted ADT to illustrate proof with the template-Z logic.

First, there is an inference rule for characteristic instantiation, which allows a variable to be replaced with a name referring to a set. This transformation uses the IC to replace all occurrences of a variable in a template formula with its substitution. For now, it is an axiom (unproved) of the template-Z logic:

[3] In fact, this is precisely the case in the ISO standard semantics of Z [13].

$$\frac{\Gamma \vdash E_1 \lessdot S \gtrdot E_2}{S \in \mathbb{U};\ (\mathcal{IP}_{\mathcal{E}}(\Gamma \vdash E_1 \lessdot S \gtrdot E_2)\{S \mapsto \text{“}S\text{”}\})}\ \textsf{[I-S]}\ [S \notin FV(\Gamma)]$$

An inference rule for Z templates for the schema calculus inference rule $\exists\ Sc$ (appendix A), is also required; given a template representation of a Z schema,

$$\lessdot Sc \gtrdot == [\ \lessdot ScD \gtrdot \mid \lessdot ScP \gtrdot\ ]$$

then the template inference rule is:[4]

$$\frac{\Gamma;\ \lessdot Sc \gtrdot == [\ \lessdot ScD \gtrdot \mid \lessdot ScP \gtrdot\ ] \vdash \exists\ \lessdot Sc \gtrdot \bullet \lessdot P \gtrdot}{\Gamma;\ \lessdot Sc \gtrdot == [\ \lessdot ScD \gtrdot \mid \lessdot ScP \gtrdot\ ] \vdash \exists \lessdot ScD \gtrdot \bullet \lessdot P \gtrdot \wedge \lessdot ScP \gtrdot}\ [\textsf{T}\ \exists\ Sc]$$

We also need the one-point rule (appendix A) for templates:

$$\frac{\Gamma \vdash \exists\ \lessdot x \gtrdot : \lessdot t \gtrdot \bullet \lessdot P \gtrdot \wedge \lessdot x \gtrdot = \lessdot v \gtrdot}{\Gamma \vdash \lessdot P \gtrdot [\ \lessdot x \gtrdot := \lessdot v \gtrdot\ ] \wedge \lessdot v \gtrdot \in \lessdot t \gtrdot}\ \textsf{[T one-point]}\ [\lessdot x \gtrdot \notin FV(\lessdot v \gtrdot)]$$

These two inference rules are easily proved by using the axiom rules of the template-Z logic and the logic of Z (see [12]).

The metaproof of, $\vdash ?\ \exists\ \lessdot P \gtrdot Init \bullet \text{true}$, using these rules is:

$\equiv$ [by $T\ \ \exists\ \ Sc$]
$\lessdot P \gtrdot == [\ \lessdot ids \gtrdot : \mathbb{P} \lessdot ID \gtrdot ;\ \lessdot st \gtrdot : \lessdot ID \gtrdot \leftrightarrow \lessdot S \gtrdot \mid$
  $\text{dom} \lessdot st \gtrdot = \lessdot ids \gtrdot \wedge \lessdot I \gtrdot\ ];$
$\lessdot P \gtrdot Init == [\ \lessdot P \gtrdot\ ' \mid \lessdot ids \gtrdot ' = \varnothing \wedge \lessdot st \gtrdot ' = \varnothing\ ]$
$\vdash \exists\ \lessdot P \gtrdot\ ' \bullet \lessdot ids \gtrdot ' = \varnothing \wedge \lessdot st \gtrdot ' = \varnothing \wedge \text{true}$
$\equiv$ [by $T\ \ \exists\ \ Sc$; sequent calculus; propositional calculus]
$\vdash \exists\ \lessdot ids \gtrdot ' : \mathbb{P}\ \lessdot ID \gtrdot ;\ \lessdot st \gtrdot ' : \lessdot ID \gtrdot \leftrightarrow \lessdot S \gtrdot \bullet$
  $\lessdot ids \gtrdot ' = \varnothing \wedge \lessdot st \gtrdot ' = \varnothing \wedge \text{dom}\ \lessdot st \gtrdot ' = \lessdot ids \gtrdot ' \wedge \lessdot I \gtrdot$
$\equiv$ [by $T\ \ one - point$]
$\vdash \varnothing \in \mathbb{P}\ \lessdot ID \gtrdot \wedge \varnothing \in \lessdot ID \gtrdot \leftrightarrow \lessdot S \gtrdot \wedge \text{dom}\ \varnothing = \varnothing$
  $\wedge \lessdot I \gtrdot ' [\lessdot ids \gtrdot ' := \varnothing, \lessdot st \gtrdot ' := \varnothing]$
$\equiv$ [by I-S twice and set theory]
$ID \in \mathbb{U};\ S \in \mathbb{U}$
$\vdash \varnothing \in \mathbb{P}\ \ ID \wedge \varnothing \in ID \leftrightarrow S \wedge \lessdot I \gtrdot ' [\lessdot ids \gtrdot ' := \varnothing, \lessdot st \gtrdot ' := \varnothing]$
$\equiv$ [by set theory and propositional calculus]
$\lessdot I \gtrdot ' [\lessdot ids \gtrdot ' := \varnothing, \lessdot st \gtrdot ' := \varnothing]$

And if I is instantiated with *true*, then the formula reduces to true.

6 Discussion

The Z language supports generic structures, but these are not a substitute for templates. Z generics only allow parameterisation with sets. This makes it impossible, for instance, to represent the templates presented here with Z generics.

[4] Template inference rules are preceeded by T.

FTL was defined formally with the aim of mechanical metaproof. However, the language became much clearer, we gained a better understanding of what templates are, what can be done with them, and of metaproof, and it also makes the construction of an FTL tool a simple extension of our work so far.

Testing the FTL semantics and IC with the Z/Eves prover helped to uncover many errors. Placeholders are simple and easy to get right, but more complex constructs (such as lists) required a lot more testing to get their definitions right. The consistency between the calculus and the semantics was tested, and a full proof of consistency was not yet done at this stage. In fact, both the IC and the actual semantics give the semantics of FTL: the actual semantics does it in terms of total instantiations, whereas the IC does it in terms of partial instantiations.

As said above, inference rules related with *characteristic instantiation* are, for now, axioms (unproved) of the template-Z logic, which would have to be proved for the logic to be claimed sound. We believe that this is proved by applications of the generalisation rule in a proper meta-world of Z (as discussed in sec. 5). metatheorems capture our experience in proving theorems with instances of some structure; they formalise something done and perceived in practice.

As the example shows, templates may need to be constrained so that useful results can be extracted. So, template design needs to consider what is to be described, and the results to be proved. The most attractive aspect of metaproof is the property *true by construction*. Often, however, metaproof gives a simplification of the original conjecture, which, in many cases, is sufficient to allow the prover to discharge the remaining formula automatically.

Metaproof has been applied to proofs that are more complex than the one presented here. For example, metatheorems for our object-oriented (OO) style for Z [6], such as, initialisation of the whole system (built as composition of components), promoted operations, and composite system operations [12].

The approach presented here was used to build a catalogue of templates and metatheorems for a framework to construct UML-based models [12, 14]. Templates capture the form of OO Z models [6], metatheorems capture model-consistency results, so that UML models are represented in Z by instantiating templates, and required consistency conjectures simplified with metatheorems.

7 Related Work

Catalysis [15] proposes templates and hints at variable substitution for instantiation, but this is defined informally. Moreover, its template notation has fewer features than FTL (Catalysis has only placeholders; FTL has lists and choice).

Patterns have been used in the setting of temporal logic [2, 3]. These works use *schematic* representations of patterns (similar to logical inference rules); instantiation is variable substituion. However, this has less abstraction contructs than FTL (placeholders only), and mechanisation is not addressed.

The approach behind FTL and the IC is akin to term-rewriting systems [16], which are methods for replacing subterms of a formula with other terms based on rewriting rules. In our approach, the FTL semantics and IC define rules for the substitution of placeholders, lists and choice. Term-rewriting is used to capture the form of objects; *L-Systems* [17], for instance, is a string rewriting system designed to capture the form of plant growth, which has many application in computer graphics. FTL captures the form of formal language sentences.

The term *template* is sometimes used to refer to generics (e.g. C++ templates), which allow parameterisation based on types, and may involve subtyping and polymorphism. Generics are more restricted than FTL templates.

8 Conclusions and Future Work

This paper presents FTL, a language to express templates, a calculus of instantiations for FTL templates and an approach that explores templates of formal models for proof. FTL allows the representation of structural patterns of modelling and proof and their mechanical instantiation. This enables reuse in formal development, which contributes to reduce the effort involved in formal modelling and verification. In our experience, the use of modelling patterns allow us to concentrate more on the problem and less on formalisation issues, and metaproof helps to reduce the proof overhead associated with formal development.

Our approach tries to address several requirements. We want to capture the form of sentences of any language, trying to separate form from content, and use templates for reasoning. So, FTL has a very general semantics, and requires further work to be integrated with some target language for reasoning. There is a separation of concerns: on one side the language to describe form, on the other the approach to reason with those representations. This constitutes a pragmatic and non-intrusive approach, rather than extending a language to support templates, we designed a general language to capture form.

Formal development requires expertise in the use of proof tools. Our approach allows experts to build templates and prove metatheorems (perhaps assisted by proof tools), so that software developers who are not experts in formal-methods can still build formal models that are proved consistent.

Future work will look at completing the proof logic for template-Z, by proving the inference rules that are now laid as axioms of the logic. We also want to add more features to FTL, such as, naming of templates and conditional instantiation. It would be interesting to apply FTL to another formal-method, and then plug a template-logic to enable formal metaproof.

This work lays the ground for tool support for FTL and metaproof. So that users can automatically generate models by instantiating templates of a catalogue, and to simplify conjectures by instantiating metatheorems. It would also be interesting to define a theory for template-Z logic in a prover, such as Proofpower [18], to enable mechanical theorem proving with template-Z.

Acknowledgements. We would like to thank Jim Woodcock for his interesting comments. This research was supported for Amálio by the Portuguese Foundation for Science and Technology under grant 6904/2001.

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Resusable Object-Oriented Software.* Professional Computing. Addison-Wesley (1995)
2. Darimont, R., van Lamsweerde, A. Formal refinement patterns for goal-driven requirements elaboration. In *SIGSOFT '96*, pp. 179–190. ACM Press (1996)
3. Dwyer, M. B., Avrunin, G. S., Corbett, J. C. Patterns in property specifications for finite-state verification. In *ICSE '99*, pp. 411–420. IEEE (1999)
4. Stepney, S., Polack, F., Toyn, I. Patterns to guide practical refactoring: examples targetting promotion in Z. In Bert, D., et al., eds., *ZB 2003, Turku, Finland*, vol. 2651 of *LNCS*, pp. 20–39. Springer (2003)
5. Abrial, J.-R. Using design patterns in formal developments. In *RefineNet, Workshop of ICFEM 2005* (2005)
6. Amálio, N., Polack, F., Stepney, S. An object-oriented structuring for Z based on views. In Treharne et al. [19], pp. 262–278
7. Amálio, N., Polack, F., Stepney, S. *Software Specification Methods: an overview using a case study*, chapter UML+Z: UML augmented with Z. Hermes Science (2006)
8. Amálio, N., Stepney, S., Polack, F. Formal proof from UML models. In Davies, J., et al., eds., *ICFEM 2004*, vol. 3308 of *LNCS*, pp. 418–433. Springer (2004)
9. Woodcock, J., Davies, J. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall (1996)
10. Saaltink, M. The Z/EVES system. In *ZUM'97*, vol. 1212 of *LNCS*. Springer (1997)
11. Tennent, R. D. The denotational semantics of programming languages. *Commun. ACM*, 19(8):437–453 (1976)
12. Amálio, N. *Rigorous model-driven development with frameworks and templates.* Ph.D. thesis, Dept of Computer Science, Univ of York (2006)
13. ISO. Information technology—Z formal specification notation—syntax, type system and semantics (2002). ISO/IEC 13568:2002, International Standard
14. Amálio, N. Frameworks based on templates for rigorous model-driven development. In Romijn, J., et al., eds., *IFM 2005, Doctoral symposium*, pp. 62–68 (2005). Technical report, University of Eindhoven, CS-05-29
15. D'Sousa, D., Wills, A. C. *Object Components and Frameworks with UML: the Catalysis approach.* Addison-Wesley (1998)
16. Bezem, M., et al. *Term Rewriting Systems.* Cambridge University Press (2003)
17. Prusinkiewicz, P., Lindenmayer, A. *The algorithmic beauty of plants.* Springer (1990)
18. Arthan, R. ProofPower. http://www.lemma-one.com/ProofPower/index/index.html
19. Treharne, H., et al., eds. *ZB 2005*, vol. 3455 of *LNCS*. Springer (2005)

A Z Inference rules [9]

$$\frac{\Gamma \vdash \exists\, Sc \bullet P}{\Gamma \vdash \exists\, ScD \bullet P \land ScP}\ [\exists\ Sc]$$

Sc is any schema; *ScD* is its declarations and *ScP* its predicate. *P* is any predicate.

$$\frac{\Gamma \vdash \exists\ x : A \bullet P \land x = v}{\Gamma \vdash P[x := v] \land v \in A}\ [\mathsf{one\text{-}point}]\ [x \notin FV(v)]$$