T U M

I N S T I T U T F Ü R I N F O R M A T I K

Theorem Proving in Higher Order Logics — Emerging Trends Proceedings

Stefan Berghofer and Makarius Wenzel (Eds.)

TUM-I0916
August 09

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-08-I0916-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Theorem Proving in Higher Order Logics
Emerging Trends Proceedings

Stefan Berghofer and Makarius Wenzel (Eds.)

August 2009, München, Germany

Abstract. This technical report is the Emerging Trends proceedings of the *22th International Conference on Theorem Proving in Higher Order Logics* (TPHOLs 2009), which was held during 17–20 August 2009 in Munich, Germany. TPHOLs covers all aspects of theorem proving in higher order logics as well as related topics in theorem proving and verification. In keeping with longstanding tradition, the Emerging Trends track of TPHOLs 2009 offered a venue for the presentation of work in progress, where researchers invited discussion by means of a brief introductory talk and then discussed their work at a poster session.

Table of Contents

Towards an Implicit Calculus of Inductive Constructions

Extending the Implicit Calculus of Constructions with Union and Subset Types

Bruno Bernardo

Projet TypiCal
Ecole Polytechnique and INRIA Saclay, France
Bruno.Bernardo@lix.polytechnique.fr

Abstract. We present extensions of Miquel's Implicit Calculus of Constructions (ICC) and Barras and Bernardo's decidable Implicit Calculus of Constructions (ICC*) with union and subset types. The purpose of these systems is to solve the problem of interaction betweeen logical and computational data. This is a work in progress and our long term goal is to add the whole inductive types to ICC and ICC* in order to define a complete framework for theorem proving.

1 Introduction

The Calculus of Inductive Constructions (CIC) [8], the formalism on which the Coq proof assistant [10] is based, is a powerful type system for theorem proving. However, one drawback of this formalism is the interaction between logical and computational subterms. Proofs, type annotations, dependencies are not computationally relevant but their content is inspected by the type checker as if they were. This can lead to a certain lack of flexibility that makes theorem proving harder and less intuitive.

Let us for example consider the euclidean division `div`. Its type is :

```
forall a b, b > 0 -> {q & {r | a = b * q + r /\ b > r}}
```

Given two integers a and b and two proofs Π_1 and Π_2 that $b > 0$, it would quite natural to consider that `div` a b Π_1 and `div` a b Π_2 are equivalent, since these programs have the same computational behaviour. However, since we do not have proof-irrelevance, it will be hard to prove that `div` a b Π_1 $=$ `div` a b Π_2 holds because Π_1 and Π_2 may differ.

In order to distinguish proofs from algorithm, Coq has two sorts Prop and Set that are almost identical regarding typing rules, but that have different purposes. Prop is intended to include types that correspond to logical propositions whereas Set is intended to include datatypes. This distinction is useful for Coq's extraction procedure [4] that removes all the logical data, or more precisely the data whose type is in Prop, and keeps the data whose type is in Set, in order to produce certified programs. Even if Coq's extraction is very powerful, it is useless during the proof elaboration stage. Moreover, since it can only erase terms whose types are in Prop, it cannot remove logical data whose type is in Set. If we consider lists indexed by their length (also known as vectors),

their index will not be erased by the extraction (since we have `nat : Set`), even though this index is a static property and is thus useless for the computation.

Miquel's Implicit Calculus of Constructions (ICC) [7] seems to be a possible solution for the interference between logical and computational parts. ICC is a Curry-style Calculus of Constructions that also have an implicit product $\forall x : T.U$ that behaves as an intersection type rather than a function type. In ICC proofs and dependencies can be implicit and thus not interfere with computational data. Moreover abstractions do not carry type annotations. If we consider again the euclidean division program `div`, the proof Π that `b>0` does not need to appear and our previous problem shall not occur : there will only be one program `div` a b whatever is the proof Π of $b > 0$ we have.

However, ICC has one major issue : it is unlikely that type inference is decidable in it. In [2], we have defined ICC*, a more verbose variant that has decidable type checking. We can see ICC* as a layer upon ICC where the implicit parts are made explicit and marked with a flag. ICC* and ICC are linked through an extraction function that removes the static part (annotations and flagged logical parts). This extraction is also used in the typing rules, mainly the conversion one: conversion is made between extracted terms. We designed ICC* so that it captures the nice behaviour of ICC while having decidable type inference.

Our long term goal is to add inductive types to both ICC and ICC* so we can have a more complete framework for theorem proving. Since inductive types can be decomposed into more basic types (sigma types, disjoint sums, fixpoint operators, equality, booleans, unit type and void type), we intend to do it by adding every basic type.

What we present here is a work in progress. First, we describe ICC$_\Sigma$, a version of ICC containing union types and subset types (Section 2), and then we present its decidable counterpart ICC$^*_\Sigma$ (Section 3).

Preliminaries In this paper we will adopt the following usual conventions. We will consider terms up to α-conversion. The set of free variables of a term t will be written $\mathsf{FV}(t)$. Arrow types are explicit non-dependent products (when $x \notin \mathsf{FV}(U)$, we write $T \to U$ for $\Pi x{:}T.\,U$). Substitution of the free occurrences of variable x by N in term M is noted $M\{x/N\}$. We will consider a set of sorts $\mathcal{S} = \{\mathsf{Prop}\} \cup \{\mathsf{Type}_i \mid i \in \mathbb{N}\}$ designating the types of types. Prop is an impredicative sort intended to represent the types of logical data whereas sorts Type_i denote the usual predicative hierarchy of the Extended Calculus of Constructions[5]. As in the traditional presentation of Pure Type Systems [1], we define two sets **Axiom** $\subset \mathcal{S}^2$ and **Rule** $\subset \mathcal{S}^3$ by

$$\begin{aligned}\textbf{Axiom} &= \{(\mathsf{Prop}, \mathsf{Type_0});\ (\mathsf{Type_i}, \mathsf{Type_{i+1}}) \mid i \in \mathbb{N}\}\\ \textbf{Rule} &= \{(\mathsf{Prop}, s, s);\ (s, \mathsf{Prop}, \mathsf{Prop}) \mid s \in \mathcal{S}\}\\ &\quad \cup \{(\mathsf{Type_i}, \mathsf{Type_j}, \mathsf{Type}_{max(i,j)}) \mid i, j, \in \mathbb{N}\}\end{aligned}$$

Note that these sets are functional and complete.

We will consider two judgements $\Gamma \vdash$ and $\Gamma \vdash M : T$. $\Gamma \vdash$ means that the context Γ is well-formed and $\Gamma \vdash M : T$ that M has type T under the context Γ.

2 ICC$_\Sigma$

ICC$_\Sigma$ is an extension of the fragment of ICC we used in [2]. Let us first recall briefly a few points about this fragment (cf. Fig1). More details about ICC's syntax can be found in [7] and [6].

2.1 Syntax of ICC

ICC is a Curry-style version of the Calculus of Constructions that also features an implicit product $\forall x : T.U$. This implicit product is interpreted as an intersection type: if we have $f : \forall x : T.U$, then we also have $f : U\{x/N\}$ for any well-typed $N : T$. (cf. rule (INST)). β- and η- reductions are defined in ICC . β-, η- and $\beta\eta$- reductions have the Church-Rosser property. Regarding typing rules, we take a restriction of the system described in [7]: we remove the (CUM),(EXT) and (STR) rules. (CUM) and (EXT) are related to subtyping, which we have not implemented yet. (STR) lets a proof of $\forall \_ : P.Q$ also be a proof of Q even if no proof of P is produced and such behaviour is not wanted. With these typing rules, $\beta\eta$ subject reduction holds. Type inference seems to be undecidable since it contains Curry-style System F, whose type inference is undecidable [11]. Coherence and strong normalization of the system are proven semantically (cf. subsection 2.3).

2.2 Adding sigma-types

In ICC$_\Sigma$, we have added a subset type $\{x : A \mid P\}$ and an union type $\exists x : A.B$ to ICC . Terms of subset type can be seen as dependent pairs where the second component (the proof that the property P holds) is implicit. Terms of union type can be seen as dependent pairs where the first component (the witness) is implicit.

There are six more typing rules (cf. Fig. 2). (SUB) and (U) are formation rules. They are very similar to the product formation rules except that every object is in the same sort s. The reason of this restriction is that the extension of the model has yet to be defined. In the near future, this restriction shall be removed.

There are two introduction rules : (SUB-I) and (U-I). (SUB-I) states that, in some context Γ, any a of type A has also type $\{x : A \mid B\}$ provided there is a proof b of type $B\{a/x\}$. (U-I) states that in some accurate context Γ, if b has type $B\{a/x\}$ with some $a : A$, then b has also type $\exists x : A.B$.

Finally elimination rules (SUB-E) and (U-E) allow us to produce an object of any sort s from an object of union or subset type.

There is no need to add any reduction rule. No projection is definable for terms of union type : the first argument is implicit so there is no first projection; since the second projection depends on the first one, we do not have a second projection either. The first projection of terms of subset type can be emulated with the (SUB-E) rule by choosing $P = \lambda x.\, A$ and $f = \lambda x.\, x$.

Church-Rosser still holds for β-, η- and $\beta\eta$- reductions : we prove it using Tait's traditional method with parallel reduction. $\beta\eta$-subject reduction does not because Tatsuta's counter-example [9] applies here (adding an implicit second-order existential to Heyting's Arithmetic breaks subject-reduction).

Sorts

$$s ::= \mathsf{Prop} \mid \mathsf{Type}_i \ (i \in \mathbb{N})$$

Terms

$$M ::= x \mid s \mid \Pi x{:}M_1.\, M_2 \mid \forall x{:}M_1.M_2 \mid \lambda x.M \mid M_1\, M_2$$

Contexts

$$\Gamma ::= [] \mid \Gamma; x : M$$

Reduction

$$(\lambda x.\, M)\, N \rhd_\beta M\{x/N\}$$
$$\lambda x.\, M\, x \rhd_\eta M \qquad (\text{if } x \notin \mathsf{FV}(M))$$

Typing rules

$$\frac{}{[] \vdash}\,(\textsc{Wf-E}) \qquad \frac{\Gamma \vdash T : s \quad x \notin \mathsf{DV}(\Gamma)}{\Gamma; x : T \vdash}\,(\textsc{Wf-S})$$

$$\frac{\Gamma \vdash \quad (s_1, s_2) \in \textbf{Axiom}}{\Gamma \vdash s_1 : s_2}\,(\textsc{Sort}) \qquad \frac{\Gamma \vdash \quad (x : T) \in \Gamma}{\Gamma \vdash x : T}\,(\textsc{Var})$$

$$\frac{\Gamma \vdash T : s_1 \quad \Gamma; x : T \vdash U : s_2 \quad (s_1, s_2, s_3) \in \textbf{Rule}}{\Gamma \vdash \Pi x{:}T.\, U : s_3}\,(\textsc{ExpProd})$$

$$\frac{\Gamma \vdash T : s_1 \quad \Gamma; x : T \vdash U : s_2 \quad (s_1, s_2, s_3) \in \textbf{Rule}}{\Gamma \vdash \forall x{:}T.U : s_3}\,(\textsc{ImpProd})$$

$$\frac{\Gamma; x : T \vdash M : U \quad \Gamma \vdash \Pi x{:}T.\, U : s}{\Gamma \vdash \lambda x.\, M : \Pi x{:}T.\, U}\,(\textsc{Lam})$$

$$\frac{\Gamma \vdash M : \Pi x{:}T.\, U \quad \Gamma \vdash N : T}{\Gamma \vdash M\, N : U\{x/N\}}\,(\textsc{App})$$

$$\frac{\Gamma; x : T \vdash M : U \quad \Gamma \vdash \forall x{:}T.U : s \quad x \notin \mathsf{FV}(M)}{\Gamma \vdash M : \forall x{:}T.U}\,(\textsc{Gen})$$

$$\frac{\Gamma \vdash M : \forall x{:}T.U \quad \Gamma \vdash N : T}{\Gamma \vdash M : U\{x/N\}}\,(\textsc{Inst})$$

$$\frac{\Gamma \vdash M : T \quad \Gamma \vdash T' : s \quad T \cong_{\beta\eta} T'}{\Gamma \vdash M : T'}\,(\textsc{Conv})$$

Fig. 1. Syntax, reduction rules and typing rules of ICC

Terms

$$M ::= \cdots \mid \{x\!:\!A \mid B\} \mid \Sigma[x\!:\!A].B$$

Typing rules

$$\frac{\Gamma \vdash A : s \qquad \Gamma; x : A \vdash B : s \qquad s \in \mathcal{S}}{\Gamma \vdash \{x\!:\!A \mid B\} : s}(\textsc{Sub})$$

$$\frac{\Gamma \vdash \{x\!:\!A \mid B\} : s \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B\{a/x\}}{\Gamma \vdash a : \{x\!:\!A \mid B\}}(\textsc{Sub-I})$$

$$\frac{\Gamma \vdash P : \{x\!:\!A \mid B\} \to s \qquad \Gamma \vdash c : \{x\!:\!A \mid B\} \qquad \Gamma \vdash f : \Pi x\!:\!A.\, \forall y\!:\!B.P\, x}{\Gamma \vdash f\, c : P\, c}(\textsc{Sub-E})$$

$$\frac{\Gamma \vdash A : s \qquad \Gamma; x : A \vdash B : s \qquad s \in \mathcal{S}}{\Gamma \vdash \exists x\!:\!A.B : s}(\text{U})$$

$$\frac{\Gamma \vdash \exists x\!:\!A.B : s \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B\{a/x\}}{\Gamma \vdash b : \exists x\!:\!A.B}(\text{U-I})$$

$$\frac{\Gamma \vdash P : \exists x\!:\!A.B \to s \qquad \Gamma \vdash c : \exists x\!:\!A.B \qquad \Gamma \vdash f : \forall x\!:\!A.\Pi y\!:\!B.\, P\, y}{\Gamma \vdash f\, c : P\, c}(\text{U-E})$$

Fig. 2. Additional syntax and typing rules of ICC_Σ

2.3 Semantics of ICC_Σ

Miquel designed in [7] two models based on coherence spaces [3]. One was used to prove that ICC is consistent; the other to prove that every well-typed term of ICC is strongly normalizing.

The extension of these models for ICC_Σ is a work in progress. Long discussions with Alexandre Miquel led us to an informal proof that ICC 's models are still valid for ICC_Σ.

In a nutshell, the interpretation of subset types is quite straightforward and follows the set theoretic intuition: $[\![\{x : A \mid B\}]\!] = \{x \in [\![A]\!] \mid [\![B]\!] \neq \emptyset\}$. Introduction and elimination rules are valid.

For union types, the argument is more intricate. The interpretation of $\Sigma[x\!:\!A].B\, x$ is the smallest semantical type that contains $\bigcup_{x \in [\![A]\!]} [\![B\, x]\!]$. Introduction and elimination rules seem valid. But the behaviour of union types is more understandable in realizability than in set theory.

3 ICC^*_Σ

ICC_Σ has two main drawbacks : subject reduction does not hold and type inference is probably undecidable. The purpose of ICC^*_Σ is to fix these problems in the same way that ICC^* fixed ICC 's type inference issue (cf. [2]). ICC^*_Σ should be seen as a wrap-

Terms

$$
\begin{array}{rl}
M ::= & x \mid s \\
& \mid \Pi x{:}M_1.\, M_2 \mid \lambda x{:}M_1.\, M_2 \mid M_1\, M_2 \qquad (explicit) \\
& \mid \Pi[x{:}M_1].\, M_2 \mid \lambda[x{:}M_1].\, M_2 \mid M_1[M_2] \quad (implicit)
\end{array}
$$

Extraction

$$
\begin{array}{cc}
s^* = s & x^* = x \\
(\Pi x{:}T.\, U)^* = \Pi x{:}T^*.\, U^* & (\Pi[x{:}T].\, U)^* = \forall x{:}T^*.U^* \\
(\lambda x{:}T.\, U)^* = \lambda x.U^* & (\lambda[x{:}T].\, U)^* = U^* \\
(M\,N)^* = M^*N^* & (M[N])^* = M^*
\end{array}
$$

Typing rules

$$\frac{}{[] \vdash}\,(\textsc{Wf-E}) \qquad \frac{\Gamma \vdash T : s \quad x \notin \mathsf{DV}(\Gamma)}{\Gamma; x : T \vdash}\,(\textsc{Wf-S})$$

$$\frac{\Gamma \vdash \quad (s_1, s_2) \in \textbf{Axiom}}{\Gamma \vdash s_1 : s_2}\,(\textsc{Sort}) \qquad \frac{\Gamma \vdash \quad (x : T) \in \Gamma}{\Gamma \vdash x : T}\,(\textsc{Var})$$

$$\frac{\Gamma \vdash T : s_1 \quad \Gamma; x : T \vdash U : s_2 \quad (s_1, s_2, s_3) \in \textbf{Rule}}{\Gamma \vdash \Pi x{:}T.\, U : s_3}\,(\textsc{E-Prod})$$

$$\frac{\Gamma \vdash T : s_1 \quad \Gamma; x : T \vdash U : s_2 \quad (s_1, s_2, s_3) \in \textbf{Rule}}{\Gamma \vdash \Pi[x{:}T].\, U : s_3}\,(\textsc{I-Prod})$$

$$\frac{\Gamma; x : T \vdash M : U \quad \Gamma \vdash \Pi x{:}T.\, U : s}{\Gamma \vdash \lambda x{:}T.\, M : \Pi x{:}T.\, U}\,(\textsc{E-Lam})$$

$$\frac{\Gamma; x : T \vdash M : U \quad \Gamma \vdash \Pi[x{:}T].\, U : s \quad x \notin \mathsf{FV}(M^*)}{\Gamma \vdash \lambda[x{:}T].\, M : \Pi[x{:}T].\, U}\,(\textsc{I-Lam})$$

$$\frac{\Gamma \vdash M : \Pi x{:}T.\, U \quad \Gamma \vdash N : T}{\Gamma \vdash M\, N : U\{x/N\}}\,(\textsc{E-App}) \qquad \frac{\Gamma \vdash M : \Pi[x{:}T].\, U \quad \Gamma \vdash N : T}{\Gamma \vdash M[N] : U\{x/N\}}\,(\textsc{I-App})$$

$$\frac{\Gamma \vdash M : T \quad \Gamma \vdash T' : s \quad T^* \cong_{\beta\eta} T'^*}{\Gamma \vdash M : T'}\,(\textsc{Conv})$$

Fig. 3. Terms, extraction and typing rules in ICC*

ping of ICC_Σ designed to capture both the semantics of ICC_Σ and the good syntactic properties of a Church-style Calculus of Constructions extended with dependent pairs. ICC^*_Σ is an extension of ICC_Σ in the same way that ICC^* is an extension of ICC.

ICC^*'s syntax is the same as in a Church-style Calculus of Constructions except that we duplicate each operation (product, abstraction and application) in an explicit one and an implicit one. In ICC^*_Σ, we add two kinds of sigma-types : $\Sigma[x:A].B$, where the first component is implicit, and $\Sigma x:A.[B]$ where the second one is implicit. The corresponding dependent pairs are respectively $([a],b)_{\Sigma[xA].B}$ and $(a,[b])_{\Sigma xA.[B]}$. We also add an elimination operator Elim (cf Fig. 4).

We defined in [2] an extraction function that removes implicit abstractions and implicit applications as well as domains of abstractions (cf. Fig 3). Here, we extend this function by removing the implicit term in dependent pairs and by mapping $\Sigma x:A.[B]$ to the subset type $\{x:A \mid B\}$ and $\Sigma[x:A].B$ to the union type $\exists x:A.B$.

Since we want ICC_Σ and ICC^*_Σ to be tightly linked, we also add six rules, each one corresponding to a rule we have added in ICC_Σ . In order to capture in ICC^*_Σ terms the behaviour of ICC_Σ terms, ICC^*_Σ rules are designed so that they match exactly ICC_Σ rules after extraction.

Moreover, regarding conversion and reduction rules, ICC^*_Σ works exactly the same way than ICC^* . This means that, in ICC^*_Σ , conversion is still made between extracted terms and not between annotated ones. Thus, there is still no need for reduction rules in ICC^*_Σ , since all the computation occurs between extracted terms.

We keep the main metatheoretical properties that we had in ICC^* . We prove by mutual structural induction that the extraction is sound and complete: for every judgement $\Gamma \vdash$ or $\Gamma \vdash M : T$ in ICC^*_Σ , $\Gamma^* \vdash$ or $\Gamma^* \vdash M^* : T^*$ holds; and vice-versa for every judgement Γ or $\Gamma \vdash M : T$ in ICC_Σ , there exists Δ or Δ, N and U such that $\Delta \vdash \wedge \Delta^* = \Gamma$ or $\Delta \vdash N : U \wedge \Delta^* = \Gamma \wedge N^* = M \wedge U^* = T$. From this, we deduce the relative consistency of ICC^*_Σ : if ICC_Σ is consistent, then ICC^*_Σ is also consistent.

We prove decidability of type inference as in ICC^* by induction using the strong normalization of ICC_Σ terms and β-reduction rules that we introduce[1] in ICC^*_Σ . Σ-types are treated the same way products are. Dependent pairs have type annotations. For the Elim operator, the induction step is straightforward.

4 Perspectives and Future Work

The final goal is to have inductive types in our system so that it could eventually be used in Coq.[2] We are going to proceed by adding each basic type that appears in the decomposition of inductive types.

The very next step is to complete the addition of Σ-types. We need to prove rigorously how Miquel's models are still valid in ICC_Σ . We shall also add explicit Σ-types $\Sigma x : A.B$, explicit dependent pairs $(a,b)_{\Sigma x:A.B}$ and allow different universes to be

[1] These rules are just defined for the sake of the proof, they play no role during computation.

[2] A prototype is already available as a darcs repository at
`http://www.lix.polytechnique.fr/Labo/Bruno.Barras/coq-implicit`

Terms

$$M ::= \cdots \mid \Sigma[x\!:\!A].B \mid \Sigma x\!:\!A.[B] \mid ([a],b)_{\Sigma[xA].B} \mid (a,[b])_{\Sigma xA.[B]} \mid \mathsf{Elim}(P,f,c)$$

Extraction

$$\begin{array}{rlrl} (\Sigma[x\!:\!A].B)^* & = \exists x\!:\!A^*.B^* & (\Sigma x\!:\!A.[B])^* & = \{x\!:\!A^* \mid B^*\} \\ (([a],b)_{\Sigma[xA].B})^* & = b^* & ((a,[b])_{\Sigma xA.[B]})^* & = a^* \end{array}$$

$$(\mathsf{Elim}(P,f,c))^* = f^*\, c^*$$

Typing rules

$$\frac{\Gamma \vdash A : s \qquad \Gamma; x : A \vdash B : s \qquad s \in \mathcal{S}}{\Gamma \vdash \Sigma x\!:\!A.[B] : s}(\Sigma_{\text{SUB}})$$

$$\frac{\Gamma \vdash \Sigma x\!:\!A.[B] : s \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B\{a/x\}}{\Gamma \vdash (a,[b])_{\Sigma xA.[B]} : \Sigma x\!:\!A.[B]}(\Sigma_{\text{SUB}}\text{-I})$$

$$\frac{\Gamma \vdash P : \Sigma x\!:\!A.[B] \to s \qquad \Gamma \vdash c : \Sigma x\!:\!A.[B] \qquad \Gamma \vdash f : \Pi x\!:\!A.\,[y\!:\!B].\,P\,(x,[y])_{\Sigma xA.[B]}}{\Gamma \vdash \mathtt{Elim}(P,f,c) : P\,c}(\Sigma_{\text{SUB}}\text{-E})$$

$$\frac{\Gamma \vdash A : s \qquad \Gamma; x : A \vdash B : s \qquad s \in \mathcal{S}}{\Gamma \vdash \Sigma[x\!:\!A].B : s}(\Sigma_{\text{U}})$$

$$\frac{\Gamma \vdash \Sigma[x\!:\!A].B : s \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B\{a/x\}}{\Gamma \vdash ([a],b)_{\Sigma[xA].B} : \Sigma[x\!:\!A].B}(\Sigma_{\text{U}}\text{-I})$$

$$\frac{\Gamma \vdash P : \Sigma[x\!:\!A].B \to s \qquad \Gamma \vdash c : \Sigma[x\!:\!A].B \qquad \Gamma \vdash f : \Pi[x\!:\!A].\,(y\!:\!B).\,P\,([x],y)_{\Sigma[xA].B}}{\Gamma \vdash \mathtt{Elim}(P,f,c) : P\,c}(\Sigma_{\text{U}}\text{-E})$$

Fig. 4. Extended syntax, extraction and typing rules in ICC^*_Σ

used in the formation rules (i.e. allow that e.g. $\Gamma \vdash \Sigma x : A.B : \mathsf{Type}_{(max(i,j))}$ when $\Gamma \vdash A : \mathsf{Type}_i$ and $\Gamma \vdash B : \mathsf{Type}_j$). These additions would require little effort with regard to syntax, but it should be harder to prove that models are still valid (or to adapt them if it is not the case).

We will then add more basic types such as unit type and void, which should be much easier than for the Σ-types.

We have already started to think about equality. The syntactic work is almost finished but the semantic part needs to be done. Our system will support heterogeneous equality, which would let us prove easily that e.g. `div 11 5 = div 13 6`, where `div` is the euclidean division.

The final step would be to add fixpoints operators so we could express recursion and have full inductive types.

Acknowledgements Many thanks to Alexandre Miquel for thorough discussions about his models of ICC and his help in extending them, to Bruno Barras for his continuous support and advice and to Mathieu Boespflug for his remarks. The author is funded by the DGA.

References

1. H. Barendregt. Lambda Calculi with Types. Technical Report 91-19, Catholic University Nijmegen, 1991. In Handbook of Logic in Computer Science, Vol II.
2. B. Barras and B. Bernardo. The implicit calculus of constructions as a programming language with dependent types. In R. M. Amadio, editor, *FoSSaCS*, volume 4962 of *Lecture Notes in Computer Science*, pages 365–379. Springer, 2008.
3. J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and types*. Cambridge University Press, New York, NY, USA, 1989.
4. P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, July 2004.
5. Z. Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.
6. A. Miquel. The implicit calculus of constructions. extending pure type systems with an intersection type binder and subtyping. In *Proceedings of the fifth International Conference on Typed Lambda Calculi and Applications (TLCA'01)*, Krakow (Poland), 2001.
7. A. Miquel. *Le Calcul des Constructions implicite: syntaxe et sémantique*. PhD thesis, Université Paris 7, Dec. 2001.
8. C. Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, Dec. 1996.
9. M. Tatsuta. Simple saturated sets for disjunction and second-order existential quantification. In S. R. D. Rocca, editor, *TLCA*, volume 4583 of *Lecture Notes in Computer Science*, pages 366–380. Springer, 2007.
10. The Coq development team. The coq proof assistant reference manual v8.2. Technical report, INRIA, France, February 2009. `http://coq.inria.fr/doc/main.html`.
11. J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Ann. Pure Appl. Logic*, 98(1–3):111–156, 1999.

Towards a Modular Extensible Isabelle Interface

Holger Gast

Wilhelm-Schickard-Institut für Informatik
University of Tübingen
gast@informatik.uni-tuebingen.de

Abstract. We present the architecture of an Isabelle user interface based on design principles and mechanisms from the Netbeans platform. The architecture addresses the challenges of maintainability and extensibility through a modular structure and the novel concept of features as units of prover functionality.

1 Introduction

Maintaining a user interface for a theorem prover can be a challenge. Since both the prover and the GUI technology are subject to frequent extensions and changes, the interface software must be adapted and extended in parallel. This problem is especially pressing as the maintenance of the interface takes resources off the central concern of developing the prover itself. Different strategies to solve this challenge have been proposed.

The first strategy is to distribute with each prover version a corresponding version of a specialized interface (e.g. [18, 14, 1]). In this solution, the burden of maintenance lies entirely with the prover developers, but the close coupling between the prover and interface also allows direct and pragmatic solutions. For instance, access to the term structure and the proof state can be implemented through shared data. In a variant, the prover interface is embedded into a programmable text editor since the high-level access to proof scripts offered there is more stable than the details of the underlying GUI libraries (e.g. [17, 2]).

A second strategy is to develop a generic interface, to specify a public protocol, and to let the prover and interface communicate only through this protocol [4, 3]. Ideally, this solution separates changes in the prover and in the interface from each other entirely. In practice, meeting the requirements of the protocol for a given prover can be surprisingly hard and may even require changes to the software structure of the prover [20]. Furthermore, designing a single protocol that anticipates the demands and capabilities of several provers is a major challenge, given that the provers change over time. However, the burden of maintenance is distributed between the developers of the user interface and the prover.

Finally, it is possible to implement advanced user interfaces with virtually no support from the prover [12]. By employing standard patterns in user interface design, the software can be structured in such a way that changes necessitated by developments in the prover can be accommodated by local changes to the interface software. The solution has, however, the drawback that the interface

software must duplicate some functionality already present in the prover. In particular, the demands on parsing proof scripts are substantial.

The conclusion to be drawn from this overview is that neither a strict separation nor a tight integration of prover and interface are ideal solutions to the maintenance problem. While the user interface should be mostly independent of the prover, it should access already implemented functionality through well-defined interfaces. When the prover is modified, most of the interface implementation should remain unchanged.

A second concern in designing the prover interface is extensibility. The user experience can often be enhanced substantially by application-specific modes of interaction (e.g. [16]). In such settings, the interface must be extended with specialized views that access functionality added to the prover in loaded theories.

This paper approaches the maintenance and extensibility problems by applying techniques used in the Netbeans platform [8]. As a platform for application programming, Netbeans provides patterns and mechanisms for assembling different modules into a consistent application. Using these mechanisms, we propose a modular architecture for an Isabelle interface. Specifically, we address the problem of translating new prover functionality into increased interface functionality: the overall functionality is split into fine-grained, self-contained *features*. Each feature comprises a prover-side implementation, an interface-side proxy [11], and one or more interface-side views that allow the user to access the functionality. The purpose of this split is to confine changes in the prover to specific features, thus reducing the part of the interface code that must be modified when the prover changes. Furthermore, new features can be installed into the prover at run-time to enable application-specific interaction modes. The concepts described in this paper have been validated through a prototype implementation.

Organization Section 2 summarizes the Netbeans mechanisms used. Section 3 describes the overall architecture. Section 4 gives applications that further motivate the decisions taken in the architecture. Section 5 concludes.

Acknowledgments I would like to thank Makarius Wenzel and Burkhart Wolff for discussions on the design goals of user interfaces for theorem provers, on the challenges encountered in the day-to-day maintenance of the Isabelle interface, and in programming interface extensions.

2 Netbeans Mechanisms for Modularity

The Netbeans platform [8], which underlies the Netbeans IDE, is a framework that supports the development of arbitrary applications. Its modular structure, platform independence, and straightforward usage make it particularly attractive for building a prover interface. The modularity itself rests on a few generic mechanisms, to be summarized in this section, which can be leveraged to the design problems identified in the introduction. Furthermore, the mechanisms described in this section are available as standalone libraries and can be used independently of the Netbeans platform itself.

2.1 Modules

A Netbeans module is a standard JAR archive that will be loaded into the platform at runtime. Its manifest may contain additional meta-information such as the version and dependencies on other modules [8, Ch. 3]. Dynamic loading and unloading are supported as well [8, §3.3]. An update mechanism [8, Ch. 22] supports the distribution of new versions of modules to end-users. The reliance on JVM mechanisms, as well as a mature IDE support, ensure that splitting an application into different modules does not lead to much development overhead, compared to a monolithic implementation.

2.2 Lookups

Frameworks in general provide generic mechanisms that can be adapted to specific application contexts. This goal requires that the generic mechanisms can access application-specific functionality. In the Netbeans platform, objects expose their specific capabilities through *lookups* [8, §5.2]. A lookup is a collection of objects that can be queried using a required type. The interface `Lookup` provides a method that returns the first object in the collection implementing the interface given by the `clazz` argument:

```
T lookup(Class<T> clazz)
```

Consider, for instance, a menu item "Save" that operates on the current selection. Netbeans represents the selection by a set of generic *nodes* [8, Ch. 9]. Since not all selected elements can be saved, the "Save" entry must access that capability, which is represented by the `SaveCookie`, through the node's lookup. If `s` is a selected element, it is sufficient to write:

```
SaveCookie sc = s.getLookup().lookup(SaveCookie.class);
```

If `sc` is not `null`, then `s` can be saved using `sc.save()`. Menu- and toolbar entries can also use the current selection's capabilities to determine whether they should be disabled [8, §5.5.].

2.3 The System Filesystem

Netbeans provides an abstraction over the OS filesystem [8, Ch. 6, 10]: files are assigned MIME types, and they can be accessed using readers registered for the MIME types. Writing applications that manipulate already defined types of files is thus greatly simplified. Besides this abstraction, the *system filesystem* also contains virtual paths contributed by the modules loaded into the platform. If a module's JAR manifest contains a reference to a `layer.xml` file, that file's content is visible in the system filesystem, and other modules can access the contribution. The system filesystem is used as the central extensibility mechanism in Netbeans.

For instance, an *action* is a UI representation of functionality. Actions usually appear in menus and toolbars, and the user can customize a Netbeans applications by assigning them to specific places. A module that wishes to contribute

a new action simply places its implementation class into a sub-folder of folder `Actions` in the system filesystem. The extension `.instance` here indicates that the file represents an instance of a Java class [8, §6.6.]. Accessing the file yields that instance.

```
<filesystem>
 <folder name="Actions">
  <folder name="Isabelle">
   <file name="org-isabelle-theoryactions-GotoPointAction.instance"/>
  </folder>
 </folder>
</filesystem>
```

A frequent access pattern is to extract from a folder all instances that implement a given interface. The convenience class `Lookups` provides a method that makes all instances in a given folder available in a lookup (Section 2.2):

```
Lookup forPath(String path)
```

They can then be retrieved using the following method in `Lookup`:

```
Collection<? extends T> lookupAll(Class<T> clazz)
```

In summary, the Netbeans platform makes it simple to write extensible applications: an extension point defines the required capabilities of extensions as an interface type; it then specifies a folder in the system filesystem where extensions should be deposited; finally, it accesses the extensions using `Lookups.forPath()`. The approach is particularly lightweight in that the main part of the work is carried out at the programming language level — it is not necessary to understand a separate extension mechanism [10].

3 Architecture

This section describes the architecture addressing the problems stated in the introduction. Section 3.1 gives a general overview. Section 3.2 treats prover-specific capabilities in more detail. Section 3.3 shows an example feature loaded into the prover at startup. Section 3.4 points out aspects concerning the maintainability.

3.1 Overview

The system is divided into three layers (Figure 1): the lowest layer contains the prover running in a separate process. The infrastructure layer contains the functionality, including the management of proof documents (cf. the model-view-controller pattern [9]). A thin presentation layer lets the user interact with proof documents and displays the current state of the proof processing. The presentation layer does not contain functionality itself, but delegates all requests to the infrastructure. For instance, the theory outline component merely implements an adapter [11] that translates the existing proof document structure into a tree data structure suitable for the Netbeans UI components.

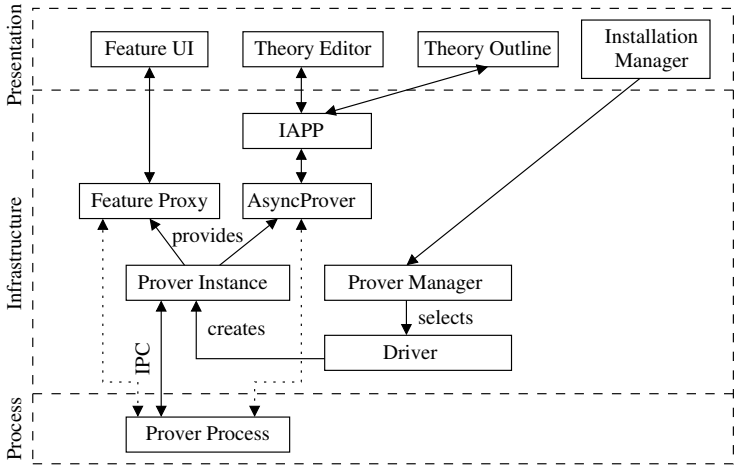

Fig. 1. Architecture Overview

Since the prover runs in a separate process, it must be accessed via inter-process communication (IPC). Following Wenzel [19], the actual communication protocol is hidden. The *prover instance* object encapsulates the life-time of a single prover process and the communication. The prover instance will usually depend on the specific prover version used.

However, the prover instance is not a monolithic object defining the interface to the prover once and forall. Instead, it exposes capabilities of the supported prover version as *features* using the lookup mechanism (Section 2.2). One feature is the support required by the IAPP (infrastructure for asynchronous proof processing) [13]. However, the set of features is not fixed (Section 3.2). The features are represented as proxies [11] that access functionality inside the prover in the background. Again, the prover communication is encapsulated.

The *prover manager* keeps track of provers installed on the system. It uses a *prover driver* to create a suitable prover instance for an installed prover. Prover drivers are implemented as separate modules and registered with the prover manager via the system filesystem (Section 2.3). This setup allows the driver for a new prover version to be distributed through the Netbeans update mechanism [8, Ch. 22]. It is expected that prover drivers can determine whether they apply to a given prover installation, for instance by checking the version number printed by the prover upon startup. The platform can thus accommodate any number of drivers simultaneously.

3.2 Features

Features represent the capabilities of specific prover versions. Figure 2 highlights the role of a feature as a remote proxy [11]: a UI component, or a component in the infrastructure, accesses the capability by a method call. In the background,

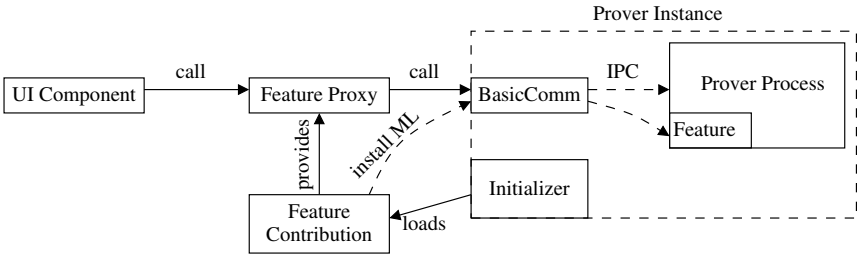

Fig. 2. Feature Proxies

the feature proxy communicates with the prover process to compute the result. Two extensions to the basic idea are worth mentioning. First, the feature proxy itself usually does not use low-level IPC. Instead, it uses a basic communication facility shared by all features in a given prover instance. This facility is, again, published as a feature (see Section 3.3).

Second, not all features will be available in the basic prover version: some may be application-specific, others may be contributed by developers of the user interface. At startup time, the prover instance therefore scans a particular directory in the system filesystem (Section 2.3) for feature implementations, identified by the interface `FeatureImpl`. Each feature implementation is given the chance to install new functionality into the running prover by sending ML code. The result of the method `installFeature()` is a new feature proxy, which is added to the prover instance's lookup. The code for loading feature extensions is, indeed, very short:

```
Lookup l = Lookups.forPath("Provers/Isabelle2008/features");
for (FeatureImpl fi: l.lookupAll(FeatureImpl.class)) {
   features.add(fi.installFeature(this));
}
```

3.3 Implementing a Contributed Feature

Contributing new functionality is straightforward. We consider as an example a feature that extracts information from the current proof state. It is prototypical of applications that wish to render goals in an application-specific fashion. A module `ExtStateQuery` (`Ext` for *extension*) contains the implementation.

First, the functionality is implemented at the ML level by defining a new (improper) Isar command "`ext_state_query` *what*", whose parameter indicates which information is to be retrieved. Using Isabelle's `Term` module, the constants in the current proof state are determined easily and written to the standard output. The ML implementation resides in `ExtStateQuery` as a plain text file.

Next, the ML implementation is installed to the prover during startup. Towards that end, `ExtStateQuery` registers the class `StateQueryImpl` with the prover driver using the `layer.xml` file (Sections 2.3, 3.2):

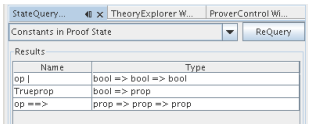

Fig. 3. View Component for State Query

```
<folder name="Provers">
 <folder name="Isabelle2008">
  <folder name="features">
   <file name="StateQuery.instance">
    <attr name="instanceClass"
      stringvalue="org.isabelle.ext.statequery.prover.StateQueryImpl"/>
   ...
```

Class `StateQueryImpl` has a method `Object installFeature(ProverInstance pi)`. It reads the ML file content into a variable and sends it to the prover using a basic communication facility:

```
InjectedCommands inj = proverInst.getFeature(InjectedCommands.class);
inj.ML(mlFileContent)
```

The method `getFeature()` is a convenience for `getLookup().lookup()`. The `InjectedCommands` feature executes commands when the prover becomes idle. The commands are thus injected into the normal processing of proof documents.[1] After installing the ML implementation in this way, the method returns the feature proxy of type of `StateQueryFeature`.

The class `StateQueryFeature` provides a method that carries out the query in the background. Its signature is `Constant[] constantsInProofState()`, i.e. it presents the result as proper objects. The method's implementation uses, again, `InjectedCommands` to communicate with the prover:

```
InjectionResult res = inj.command("ext_state_query constants");
```

The `InjectionResult` is a future, a placeholder for a result computed asynchronously. Using `res.await()`, the method blocks until the prover has executed the command. The future then contains the standard output of `ext_state_query`, which is analyzed and packed into `Constant` objects (with name and type).

Finally, a view using the feature is implemented directly using Netbeans' Matisse GUI builder (Figure 3). Accessing the constants in the current goals in its implementation is straightforward, using the proxy introduced above:

```
StateQueryFeature q = pi.getFeature(StateQueryFeature.class);
Constant res[] = q.constantsInProofState();
```

[1] For asynchronous proof processing[13], `command()` takes an additional parameter, the ID of the command after which the injected command should be executed.

3.4 Addressing Maintainability

The architecture supports maintainability by modularity: features represent fine-grained units of functionality, made available to the user interface through standard method calls. The remainder of the infrastructure and presentation layers is insulated from changes to the feature implementation as long as the method-call interface remains the same. Furthermore, it can be expected that changes to one feature do not affect other features. The result is that changes to the prover have local effects on a well-defined parts of the interface implementation.

Furthermore, the strict division of presentation and infrastructure makes the infrastructure testable by standard approaches such as JUnit. Testability, on the other hand, is a prerequisite for changeability and maintainability [6]: developers can be more confident about necessary changes if they can determine quickly and automatically whether existing functionality has been broken. Note that the situation with a user interface differs from the case for the prover itself: a change has not broken a prover's functionality if all libraries continue to be accepted. Most code in a user interface, on the other hand, is executed only very infrequently, when the user happens to request a particular action.

4 Further Applications

The proposed architecture supports several usages beyond a standard prover interface. This section sketches these applications briefly.

Isabelle as a Back-end Prover Isabelle is not only used for interactive proof development, but also for background proofs in higher-order logic (e.g. [21]). In such applications, the infrastructure layer can serve as a high-level interface to access Isabelle's functionality. In particular, the `InjectedCommands` feature (Section 3.3) enables execution of single commands.

Background Queries The example in Section 3.3 shows how the term structure in goal states can be queried through features. We expect this possibility to be very useful for application-specific theorem proving where theories would be accompanied by specialized viewers for the formalized objects. Proof-by-pointing applications [16, 15] could be realized by suitable markups in terms. More extensive computations, for instance the generation of fragments of proof documents [5], could be realized on the prover side by contributed features as well.

Driver-specific Capabilities Prover drivers so far are responsible only for the creation of prover instances for installed provers. However, by applying the lookup pattern to the drivers themselves, more detailed support can be made available. For instance, an `InstallationFeature` might offer the capability of downloading and installing the prover itself into a particular location and configuring and compiling it according to the system environment.[2]

[2] Thanks to Burkhart Wolff for this suggestion.

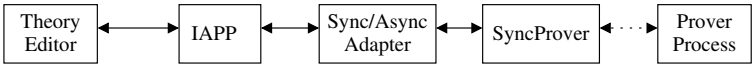

Fig. 4. Support for Synchronous Provers

Synchronous Provers The IAPP, as defined in [13], assumes the prover instance to support the protocol for asynchronous proof processing, even if the proving proceeds in linear order effectively. For backwards compatibility, it would be desirable if a prover could implement a simpler `SyncProver` interface, modelled e.g. after [7]. Using features, the desired adaptation is straightforward (Figure 4): when failing to retrieve the `AsyncProver` feature, the IAPP asks for the `SyncProver` feature and uses a simple adapter, the emulator component from [13, Figure 5], to obtain the desired functionality.

5 Conclusion

We have proposed a modular architecture for an Isabelle user interface. Its main design goal is to simplify the maintenance of the prover interface. Towards that end, we have introduced *features* as units of functionality that consist of a prover-side implementation and a interface-side proxy. The proxy object hides the background communication such that changes to the prover do not affect the interface. The architecture separates an infrastructure layer, which provides the functionality, from a presentation layer, which is responsible for the user interaction. The division increases testability as a necessary prerequisite to changeability, and makes Isabelle accessible as a background prover in non-interactive contexts. Finally, the architecture supports application-specific theorem proving by contributed features, which access functionality that is related to loaded theories or is installed at startup.

The design elements underlying the architecture are taken from the Netbeans platform: its *lookups* are used to publish the specific capabilities of objects and its *system filesystem* serves as an extension mechanism. Since these facilities are also available as standalone libraries, the architecture does not, in principle, depend on a Netbeans realization of the entire interface.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4(1):32–54, 2005.
2. D. Aspinall. Proof General: A generic tool for proof development. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '00)*, number 1785 in LNCS, 2000.
3. D. Aspinall, C. Lüth, and A. Fayyaz. Proof General in Eclipse: System and architecture overview. In *Eclipse Technology Exchange Workshop at OOPSLA 2006*, 2006.

4. D. Aspinall, C. Lüth, and D. Winterstein. Parsing, editing, proving: the PGIP display protocol. In *International Workshop on User Interfaces for Theorem Provers 2005 (UITP'05)*, 2005.
5. D. Aspinall, C. Lüth, and B. Wolff. Assisted proof document authoring. In *Mathematical Knowledge Management 2005 (MKM '05)*, number 3863 in Springer LNAI, pages 65–80, 2005.
6. K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman, Amsterdam, 1999.
7. Y. Bertot and L. Théry. A generic approach to building user interfaces for theorem provers. *J. Symbolic Computation*, 25:161–194, 1998.
8. T. Boudreau, J. Tulach, and G. Wielenga. *Rich Client Programming: Plugging into the Netbeans platform*. Prentice Hall, 2007.
9. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture: A System of Patterns*, volume 1. Wiley & Sons, 1996.
10. The Eclipse workbench. http://www.eclipse.org.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.
12. H. Gast. An architecture for extensible Click'n Prove interfaces. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics: Emerging Trends Proceedings*, number 364/07. Department of Computer Science, University of Kaiserslautern, Aug. 2007.
13. H. Gast. Managing proof documents for asynchronous processing. In *User Interfaces for Theorem Provers (UITPs 2008)*, volume 226 of *ENTCS*, pages 49–66. Elsevier Science Publishers B. V., 2009.
14. D. Haneberg, S. Bäumler, M. Balser, H. Grandy, F. Ortmeier, W. Reif, G. Schellhorn, J. Schmitt, and K. Stenzel. The user interface of the KIV verification system - a system description. In *Proceedings of the User Interfaces for Theorem Provers Workshop (UITP 2005)*, 2005.
15. C. Lüth, H. Tej, Kolyang, and B. Krieg-Brückner. TAS and IsaWin: Tools for transformational program development and theorem proving. In *Fundamental Approaches to Software Engineering: Second International Conference (FASE'99)*, volume 1577 of *LNCS*, 1999.
16. C. Lüth and B. Wolff. Functional design and implementation of graphical user interfaces for theorem provers. *Journal of Functional Programming*, 19(2):167–189, 1999.
17. S. Owre. A brief overview of the PVS user interface (invited tutorial). In *UITP*, 2008.
18. C. Team. The Coq proof assistent. http://www.lix.polytechnique.fr/coq/, 2009.
19. M. Wenzel. Interactive proof documents – theorem provers for user interfaces. http://www4.in.tum.de/ wenzelm/papers/edinburgh2008.pdf, Nov. 2008.
20. M. Wenzel. personal communication, 2008.
21. K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. *SIGPLAN Not.*, 43(6):349–361, 2008.

A Mechanized Theory of Aspects

Florian Kammüller and Henry Sudhof

Technische Universität Berlin

Abstract. Aspect-orientation is a novel concept in software development and is widely recognized as an important extension of object orientation. The approach promises to overcome limitations of the plain object oriented paradigm by adding the means to modularize cross-cutting concerns. In this paper, we present our development of a theory of aspects, in the form of an extension of the ς–calculus entirely formalized in Isabelle/HOL [1]. With this model, we want to answer essential questions regarding the power and safety of aspects with a strong base of rigorous proofs.

1 Introduction

Aspect oriented programming promises to add flexibility into existing programming paradigms to allow the modularization of cross-cutting concerns. The extension of existing languages by adding a new family of constructs opens new issues and situations, which need careful examination. In particular, the static typing of aspects is less strict than in languages like java, leading to uncaught runtime errors.

Thus, the field is an ideal candidate for using an interactive theorem prover, as establishing the fundamental implications and limitations calls for a rigorous approach. For this reason, we have developed our entire approach in the theorem prover Isabelle/HOL. At the same time, we strove to make the approach as small and simple as possible, while still capable of capturing the fundamental issues. For this reason, we have decided to use the object oriented ς-calculus as basis for our aspect oriented approach.

In this paper, we will first introduce the basic concepts of aspect orientation in Section 1.1. Then we present the Isabelle formalization of the core calculus and aspects in the Sections 2.1 and 2.2, as well as the approach to typing in Section 2.3. Finally we present a discussion introducing an early concept for classifying the aspects we were able to accommodate and to establish a comparison to related work.

1.1 Aspect orientation: a brief introduction

Aspect oriented programming (aop) [9] is a novel paradigm to facilitate modularization and flexibility in software development by providing a better separation

[1] The formalization can be found on our homepage at `http://user.cs.tu-berlin.de/~flokam/ascot/`.

of concerns. It is often defined [7] as adding a new type of modules – "aspects" – to a language. Aspects are the combination of two new abilities in programming and design: the first being able to quantify over a program, the second being able to alter its behaviour without any explicit changes in the original program – called "base". This is referred to as "Quantification and Obliviousness"[7].

Aspects are themselves modules combining pairs of pointcuts and advice. Pointcuts quantify over the base program. In aop languages like AspectJ, the pointcuts are expressions in a logical pointcut-language which select situations in the base program. The situations in a program that these pointcuts quantify over are called joinpoints or exposed joinpoints; usually method invocations and executions are examples for joinpoints.

Pointcut expressions are assembled from string with wildcards that method names, i.e. purely lexicographic . These primitive lexicographic patterns are can be used in a number of advanced operators. The following example describes a pointcut that selects all invocations of methods that have names starting with `move` on instances of the class `Point`.

```
call(void Point.move*(..))
call(void Point.move*(..)) && cflow(call(void Rectangle.move*(..)))
```

The second line adds a condition that only those invocations happening in the context of an invocation to a method starting with `move` of a class named `Rectangle` should be selected.

Pointcuts alone do not alter anything; here, the other part of an aspect comes in: the advice, which is the code that details what should happen in the program areas matched by the pointcut. Technically, there, advice can either replace the original code or add code either before or after the joinpoint. However, as replacing advice can simulate all other kinds, we only consider replacing advice in this paper.

```
around (call(Object Point.move*(..)))
       return null;
```

The advice above replaces all calls to methods captured by its pointcut, i.e. all calls to methods of the class `Point` with a name starting with `move` by a null pointer. Together, advice and pointcut form the basic aspect construct: a pointcut that quantifies over a base program and advice code that acts at these places in the base program. The combination of aspects and base programs is called weaving.

Aspects are not inherently safe, as aop adds a new dimension of flexibility, which calls for careful examination of the static safety implications – not in isolation, but in combination with object oriented concepts. For instance, consider the example given in Figure 1. In the example, an aspect replaces the method `test` of the class `Point` and its subclasses. More literally, the pointcut in the aspect `Asp` states that the method `test` should be replaced in all subclasses of `Point`, indicated by the `+`. However, the subclass `ColoredPoint` re-defines that method with a more special return type. The aspect's return value – `Point` – is thus not conform to the type of the redefined method – `ColoredPoint`. The application of the aspect at all points indicated by its pointcut results in a type-error, when

the method is called in a situation expecting an instance of `ColoredPoint` to be returned. This situation constitutes a clash between the flexibility provided by the aspect and existing flexibility provided by the base-language's – here Java – concept of inheritance with contra-variant re-definition.

```
public class Point {
  public Point test() {
    return this;
  }
}
```

```
public class ColoredPoint extends Point {
  public ColoredPoint test() {
    return this;
  }
}
```

```
public aspect Asp {
  Object around(): call(* Point+.test(..)) {
    return new Point();
  }
}
```

Fig. 1. A covariance situation in AspectJ

Considering the current state of the art, we believe it to be worthwhile to establish a rigorous approach for aop.

2 Formalization

2.1 Semantics of the core calculus

The Theory of Objects consists in various ς-calculi that are aimed to be as "simple and fruitful as λ-calculi" [2]. The ς-calculus takes objects as primitive. What we call parametrized ς-calculus is a slight extension of ς-calculus with a second parameter for methods.

The actual reduction semantics is realized as a small-step operational semantics, using Isabelle's support for the inductive definition of predicates. Using this powerful Isabelle feature, allows a representation similar to the notation one would use on paper.

In the particular case of the operational semantics, we use the inductive feature to define a relation between two terms. If all the preconditions of a rule are met, then the two terms in the conclusion are in the reduction relation, i.e. the first term reduces to the second in one step.

The first rule – Beta – is of particular importance. It expresses method invocation. The substitution `[(Obj f T), b/0]` replaces the self parameter for the outermost variable – which always has the index 0 – in the object's *lth* field `f l`. At the same time, the parameter variable of the method `f l` – this parameter again with the index 0 – is replaced with the call's second term. We realize substitution by an Isabelle function over `dB`-terms with the mixfix-notation `x[y,z/0]`, which replaces `Self 0` with `y` and `Param 0` with `z` in `x`. The principle to represent function application by substitution is, in the following section, again used for weaving aspects. The rule for update simply replaces the old term with the new one in the resulting object. The next four rules: SelL, SelR, UpdL, UpdR and Obj are context rules, adding the reduction

$$\textsc{Beta}\ \frac{l \in \mathbf{dom}\ \ f}{\texttt{Call}\ (\texttt{Obj}\ f\ T)\ l\ b \rightarrow_\varsigma the(f\ l)[(\texttt{Obj}\ f\ T), b/0]} \qquad \textsc{Upd}\ \frac{l \in \mathbf{dom}\ \ f \qquad T}{\texttt{Upd}\ (\texttt{Obj}\ f\ T)\ l\ a \rightarrow_\varsigma \texttt{Obj}\ (f\ (l \mapsto a))\ T}$$

$$\textsc{SelL}\ \frac{s \rightarrow_\varsigma t}{\texttt{Call}\ s\ l\ b \rightarrow_\varsigma \texttt{Call}\ t\ l\ b} \qquad \textsc{SelR}\ \frac{s \rightarrow_\varsigma t}{\texttt{Call}\ a\ l\ s \rightarrow_\varsigma \texttt{Call}\ a\ l\ t} \qquad \textsc{UpdL}\ \frac{s \rightarrow_\varsigma t}{\texttt{Upd}\ s\ l\ u \rightarrow_\varsigma \texttt{Upd}\ t\ l\ u}$$

$$\textsc{UpdR}\ \frac{s \rightarrow_\varsigma t}{\texttt{Upd}\ u\ l\ s \rightarrow_\varsigma \texttt{Upd}\ u\ l\ t} \qquad \textsc{Obj}\ \frac{s \rightarrow_\varsigma t \qquad l \in \mathbf{dom}\ \ f}{\texttt{Obj}\ (f(l \mapsto s))\ T \rightarrow_\varsigma \texttt{Obj}\ (f(l \mapsto t))\ T} \qquad \textsc{Asp}\ \frac{s \rightarrow_\varsigma t}{l\langle s \rangle \rightarrow_\varsigma l\langle t \rangle}$$

Fig. 2. The inductive definition of the reduction relation.

of subterms inside objects and statements. The final rule allows the reduction inside labelled terms.

2.2 Aspect-orientation with ς

The aspect oriented expression of the calculus is based on using explicit labels in terms, which denote joinpoints. These labels are part of the datatype presented in the preceding section and do not have any semantics on their own. The semantics for the aforementioned labels are later added by the process of weaving. Our idea of weaving is a primitive recursive function that traverses the entire term and applies the advice when it encounters a label matching the pointcut; after applying all aspects, the labels are erased from the terms. Applying an aspect is done on term level; the advice code's self variable is replaced with the labelled expression by using the normal substitution.

This is realized by the definition of our labels: Each label wraps the term it marks and has a value acting as identifier. The actual aspects consist of two parts: one list of label identifiers and a term. The list part correspond to the pointcut and the term to the advice used in aop. The label list constitutes a quantification about labels; all labels having an identifier listed in the pointcut are considered to be joinpoints where the aspect should act, i.e. which are evaluated by the weave function. The term part – advice – is woven into the base term by replacing the using the wrapped subterms from the labels and making them available to the advice by substituting a base variable with the original subterm.

2.3 Basic typing

The typing and subtyping relations are realized as inductive predicates, as shown in Figure 3, using Isabelle's inductive package. The subtyping relation allows the contravariant re-definition of methods, with the added variation of using an annotation to safely alter between allowing the update operation and subtyping. The actual subtyping rule is Sub-Obj; the remaining two rules add the transitive

and reflexive closure. The relation is a valid partial order, a basic sanity condition for subtyping relations.

$$\textsc{Sub-Obj}\quad \frac{\forall l \in dom(B) \qquad \texttt{return}(A_l) <: \texttt{return}(B_l) \qquad \texttt{variance}(A_l) \rightarrow (\texttt{return}(A_l) = \texttt{return}(B_l) \wedge \texttt{param}(A_l) = \texttt{param}(B_l)) \qquad \texttt{param}(B_l) <: \texttt{param}(A_l) \qquad \texttt{variance}(A_l) \rightarrow \texttt{variance}(B_l)}{A <: B}$$

$$\textsc{Sub-Trans}\quad \frac{A <: B \qquad B <: C}{A <: C} \qquad\qquad \textsc{Sub-Refl}\quad \frac{}{A <: A}$$

$$\textsc{VarSelf}\quad \frac{x < |E| \qquad E{\triangleleft}x = B \qquad B <: A}{E, L \vdash \texttt{Var}\ (\texttt{Self}\ x) : A} \qquad \textsc{VarParam}\quad \frac{x < |E| \qquad E{\triangleright}x = B \qquad B <: A}{E, L \vdash \texttt{Var}\ (\texttt{Param}\ x) : A}$$

$$\textsc{Obj}\quad \frac{\textbf{dom}\ b = \textbf{dom}\ B \qquad \forall i \in \textbf{dom}\ B \qquad E\langle 0 : B, \texttt{param}(B_i)\rangle, L \vdash b_i : \texttt{return}(B_i) \qquad B <: A}{E, L \vdash \texttt{Obj}\ b\ B : A}$$

$$\textsc{Upd}\quad \frac{E, L \vdash a : B \qquad l \in \textbf{dom}\ B \qquad E\langle 0 : B, \texttt{param}(B_l)\rangle, L \vdash n : \texttt{return}(B_l) \qquad B <: A \qquad \neg\texttt{variance}(B_l)}{E, L \vdash \texttt{Upd}\ a\ l\ n : A}$$

$$\textsc{Call}\quad \frac{E, L \vdash a : B \qquad l \in \textbf{dom}\ B \qquad \texttt{return}(B_l) <: A \qquad E, L \vdash b : \texttt{param}(B_l)}{E, L \vdash \texttt{Call}\ a\ l\ b : A}$$

$$\textsc{Lab}\quad \frac{E, L \vdash a : B \qquad i < |L| \qquad L_i = B \qquad B <: A}{E, L \vdash i\langle a\rangle : A}$$

Fig. 3. The inductive definition of the subtype and typing relations.

The typing rules are very straightforwardly interpreted as follows.

- The VAR rules for typing variables are very similar to each other, only differing in the type of the variable. The rule VARSELF is for typing instances of "Self" variables, i.e. a nesting object's values. VARPARAM on the other hand is used to type method parameters in the method body. This difference is visible by the use of $\triangleleft$ and $\triangleright$ respectively. Both rules are read as "if the variable is bound then the expression's type is the one referenced in the variable environment for that variable" – all bound variables have types associated with them in the environment.
- The CALL rule for typing method invocations asserts that all sub-expressions are valid for the invocation. This involves the callee being well-typed and having the method being invoked. It also enforces that the parameter is well-typed and conforms to the parameter type for the method in the callee's type. If so, the resulting type is the one found as return type in the callee's type for the particular method.
- The OBJ rule for objects is notable, as we enforce conformity with the type annotation in the rule, linking syntax and typing. The actual rule also states that all methods of the object have to be well-typed under the assumption that the self variable 0 – the object's *self* variable – is of the object's type.

- Similarly, the object update UPD rule asserts that the object to be updated is well typed and that the method/field to be updated is defined in its type. Moreover, the new value for the method or field has to be well typed assuming self 0 to be of the updated object's type when supplying a valid parameter. The fundamental change to the update rule was that it now requires the variance annotation for the method to be updated to be false.
- To type labels, we introduced a new environment L, which maps labels to types. LAB, the corresponding typing judgement, enforces that a given label may only be used to mark terms of a given type. Label environments act as interface between aspects and base terms.

2.4 Typing of aspects

Following the realization of aspects as a non-invasive extension to the base calculus, we have also formalized the typing of aspects as an extension to the basic type system. We see the major feature of this approach in its simplicity and the nearly complete decoupling of aspects from base in the definition of weaving and in the type system. This in turn allows us to treat aspects as independent modules while still being able to guarantee static type-safety. This modularity is also reflected in the proofs: the proofs of type safety reduce to a theorem that shows that weaving preserves types [10].

The typing of aspects is realized by a condition `wf_adv`, which is defined as follows:

$$\texttt{wf\_adv}\ \mathtt{L_T}\ \langle \texttt{pc, adv}\rangle \equiv_{df} \\ \forall\ \texttt{l}\in\mathbf{dom}\ \ \mathtt{L_T}.\ []\langle \texttt{0: } \mathtt{L_T}\ \texttt{!l,void}\rangle, \mathtt{L_T} \vdash \texttt{adv: } \mathtt{L_T}\texttt{!l}$$

It states that an aspect has to be well-typed respective to an interface L. More precisely, it expresses that for each label in the aspect's pointcut, the advice is of the type listed in the environment for that label, if the aspect's self parameter was of that type. The function application is indicated by assuming 0 to be of the type given by the label environment. `void` is the empty type, indicating that the parameter is not used. This guarantees that an aspect's replacement of the original labelled expression conforms to the same type.

In other words, the result of weaving an advice has to conserve the type under the label. A notable benefit of this approach is that the condition is not based on a particular base-term, but is entirely based on $\mathtt{L_T}$, which can be seen as an generic interface. Another added benefit is the indirect instantiation of the label type using all labels in the pointcut via $\mathtt{L_T}$. This allows polymorphism of aspects, as the typing is checked against a set of types. this makes – for instance – the application of aspects without base effects possible on all marked terms. As the condition `wf_adv` is part of the typing constraints, we also prove the decidability (using a primitive recursive predicate) to guarantee static aspect typability.

Subtypes add themselves naturally to this definition because subsumption implicitly affects the typing predicate for aspects. Thus we were able to prove the soundness of weaving and aspects without altering their essential semantics. We thus are able to statically describe a strong and modular typing definition for aspects without requiring a concrete base program.

2.5 Type safety

For showing the type safety of the typing relation, we used the classical approach [14] of proving progress and preservation. In a second step, we are able to prove that weaving of well-typed aspects always yields well-typed programs. Using our strong definition of aspect typing from section 2.4, we can lift the proofs of basic type safety to aspects. This means, that both type-safety theorems stay valid, when well-typed aspects are woven. More importantly, we are able to prove static type-safety for our typing of aspects.

Theorem 1 (Aspect preservation).

$$[\![\ \texttt{wf\_adv L A; t} \Downarrow \texttt{A} \rightarrow_\varsigma \texttt{t'}\ ]\!] \Longrightarrow \texttt{e, L} \vdash\ \texttt{t': T}$$

Theorem 2 (Aspect progress).

$$[\![\ \texttt{wf\_adv L A;}\ \neg(\exists\ \texttt{c T . delabel (t} \Downarrow \texttt{A) = Obj c T})]\!] \Longrightarrow (\exists\ \texttt{t'. delabel (t} \Downarrow \texttt{A)} \rightarrow_\varsigma \texttt{t'})$$

Summarizing, we presented a modular concept for typing aspects. The simple link over a shared type dictionary is powerful enough to safely type aspects and base programs in a simple fashion. Moreover, the proofs are highly modular themselves and only require minimal changes to allow for subtyping in and with aspects.

2.6 Compositionality

We were able to deduce a notion compositionality of aspects [12]. By compositionality we mean that aspect weaving respects the evaluation of a term.

Theorem 3 (Aspect compositionality).

$$[\![\ \texttt{wf\_adv L A; t} \rightarrow_\varsigma \texttt{t'}\ ]\!] \Longrightarrow \texttt{t}\Downarrow\texttt{A} \rightarrow_\varsigma^* \texttt{t'}\Downarrow\texttt{A}$$

The theorem only needs one well-formedness conditions as hypothesis: an advice must not contain any free variables (apart from its parameter). This well-formedness is contained in the typing rules for aspects and thus – in turn – in `wf_adv` - which we chose as a common hypothesis for the above theorem.

The wider implication of this strong property of compositionality for the classification of aspects is an interesting open question. Clearly, there are many aspects expressible event based aspect models that are not compositional.

3 Related Work

The formal analysis of aspect-orientation is also being pursued by a number of other approaches. Some of these approaches use so-called labels to mark areas of the base program where aspects might be applied. Others follow a strict interpretation of the *obliviousness* property and avoid annotations in base programs entirely. There are real-life examples for the use of labels, ranging from pointcut interfaces to automatically introduced labels to resolve pointcuts [3].

Ligatti et al [13] presented an aspect-aware λ-calculus to answer type theoretic questions. The basic formalism is the simply typed λ-calculus, with added constructs like `if` and `print`. This approach uses explicit labels in the base application on which to apply aspects. These labels can be added dynamically. Aspects are handled as part of the base terms, i.e. advice and pointcuts are notated in the same expression as the base program. Pointcuts are modelled by expressions about the aforementioned labels; advice uses itself the basic calculus. Aspects can be added in any order, allowing to affect the advice ordering. Being based on the λ-calculus, there is no native support for modelling objects. A strong case for the calculus is the inclusion of a type-preserving compilation to a fully-fledged language.

A dialect of the minimal calculus by Ligatti et al [13] with a more pointed intention is Harmless Advice [5] by Dantas and Walker. This approach uses the basic calculus as described above, but adds a control-flow security type system that enforces non-interference. The calculus was significantly streamlined, removing the finer weaving semantics and adding a pseudo-imperative concatenation operator. The type system enforces strict separation between domains, guaranteeing non-interference between base and advice code.

Clifton and Leavens developed the MiniMAO series of core calculi for AspectJ and its extension MAO, using an imperative Featherweight Java dialect. The approach implements a nominal type system and avoids using labels, making it significantly different from those mentioned before. Other unique features include its ability to capture dynamic pointcuts and proceed statements. Beyond the focus on pointcut semantics there is also a soft non-compositional concept of ownership encoded into the type system.

Riley and Jagadeesan also employed a Featherweight Java dialect to show that generic types can be used as means to achieve type safety for aspects. Combining parametric object-orientation and aspects, the work presents a class–based nominal approach for type-safe aspects. The paper mentions cases where the use of generics lead to better re-usability in generic and non-generic programs alike. Even more interesting are the described situations where the use of generics leads to ways for typing otherwise unsound situations.

De Fraine, Südholt, and Jonckers contributed strongAspectJ [6], a type-safe dialect of AspectJ. By adding dual joinpoint/proceed signatures this approach proposes a type sound extension to AspectJ. These dual interfaces consist of a *proceeds* interface detailing the type expected at the join point by the aspect and an advice interface declaring what type the advice will send to the caller. By matching the interfaces against the base application, type safety is achieved. The approach removes some flexibility from AspectJ, notably enforcing that the number of arguments of the proceed statement matches that of the joinpoint. The results of this work have not been proved with the use of a theorem prover.

4 Discussion

We presented a simple, compact and powerful calculus for aspect orientation. The calculus is object oriented and entirely functional, making it easy to understand and to use. The mechanization of the calculus adds an unique foundation to our approach, that allows the rigorous study of aspects and the extraction of code, for instance type checkers or interpreters. To our knowledge, there is no other mechanized calculus with aspect orientation. We share the basic approach of using labels with Ligatti et al [13], but have added a truly object oriented setting, without losing the compactness that makes a core calculus powerful. The use of labels, just as in our work is not a drawback, as labels facilitate the reasoning about aspects by providing explicit and enumerable joinpoints. Moreover, labels can easily simulate lexicographic patterns.

4.1 Classification of aspects

Our aspect calculus is able to capture a wide range of real-world aspects. The approach of using a variable to represent the base term, allows the easy simulation of before/after/replace aspects; labels can be automatically inserted and are able to represent almost any static pointcut. Dynamic pointcuts are using additional conditions that go beyond the simple matching of pointcuts. The most important class of such dynamic pointcuts are those, which use cflow expressions, i.e. which apply only when called from the context – control flow – of certain other joinpoints. It is possible – generally [3] and in our formalization – to select labels in a way that captures some of those dynamic conditions. A trivial example – the possible scenarios are not limited to such simple cases – would be, when a method is only ever called in the context of another method.

Furthermore, it can be argued that the addition of a dynamic condition cannot make a safe aspect unsafe – dynamic conditions can only limit static conditions further and not select additional joinpoints.

The class of aspects that pass our well-formedness condition is far more selective; these aspects are known to be safe and compositional. Hence, we can classify compositional aspects as the most restrictive kind. This is followed by aspects that can be expressed in a functional calculus; we are positive that ς_{ASC} is able to capture nearly all of those aspects. For instance, we can use an aspect to count the invocations of a given labelled method. However, as the calculus is functional, it is not possible to have a global counter for all invocations of a given aspect, as that would require a reference semantic. We can thus classify aspects into compositional, functional and reference–based and are able to model the former two classes in our calculus.

4.2 Conclusion

The presented aspect calculus benefits by no small degree from being mechanized in Isabelle/HOL. The theories are modular, new steps can be tested with only replacing a few building blocks. As examples, we are currently experimenting

with a locally nameless formalization for binders and are extending the ability to handle dynamic pointcuts. Already without these future steps, the calculus is able to show the basic issues that arise when introducing aspects and means to use safe aspects. It is a compact, simple and rigorously proven concept that shows the basic limits and type-theoretic issues that arise when introducing aspects to object-orientation, especially in conjunction with width and depth subtyping. At the same time, we have the simplicity that guarantees easy extendibility to handle advanced scenarios, like distributed and parallel architectures, including component systems.

References

1. B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, S. Weirich. Engineering Formal Metatheory. In *POPL'08, ACM Press* 2008.
2. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.
3. P. Avgustinov, E. Hajiyev, N. Ongkingco, O. de Moor, D. Sereni, J. Tibble, and M. Verbaere. Semantics of static pointcuts in AspectJ. In *POPL'07, ACM Press*, 2007.
4. C. Clifton, G. T. Leavens, and J. Noble. Mao: Ownership and effects for more effective reasoning about aspects. In *ECOOP 07, Vol. **4609***, LNCS, 2007.
5. D. S. Dantas and D. Walker. Harmless advice. In *POPL '06*, pages 383–396. ACM, 2006.
6. B. De Fraine, M. Südholt, and V. Jonckers StrongAspectJ: Flexible and Safe Pointcut/Advice Bindings. AOSD 2008, ACM Press, 2008.
7. R. Filman and D. Friedman. Aspect–oriented programming is quantification and obliviousness. In *Advanced Separation of Concerns, OOPSLA*, 2000.
8. R. Jagadeesan, A. Jeffrey, and J. Riely. Typed parametric polymorphism for aspects. Sci. of Comp. Programming, 63(3):267–296, 2006.
9. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP 1997*. Vol. **1241**, LNCS, Springer, 2008.
10. F. Kammüller and H. Sudhof. *A Mechanized Framework for Aspects in Isabelle/HOL*. ACM SIGPLAN Workshop on Mechanizing Metatheory, 2007.
11. F. Kammüller and H. Sudhof. Composing safely – a type system for aspects. In *Software Composition, SC'08*. Vol. **4954**, LNCS, Springer, 2008.
12. F. Kammüller and H. Sudhof. Compositionality of aspect weaving. In *Autonomous Systems – Self Organization, Management, and Control*. Springer, 2008.
13. J. Ligatti, D. Walker, and S. Zdancewic. A type–theoretic interpretation of pointcuts and advice. Sci. of Comp. Programming, 63(3):240–266, 2006.
14. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.
15. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, LNCS **2283**, Springer, 2002.
16. The POPLmark challenge. `http://alliance.seas.upenn.edu/ ~plclub/ cgi-bin/ poplmark`. July 2007.
17. Christian Urban and Christine Tasson. Nominal Techniques in Isabelle/HOL. In *CADE 05*. LNCS, **3632**, Springer, 2005.

A Formalization of the Semantics of Functional-Logic Programming in Isabelle*

Francisco J. López-Fraguas[1], Stephan Merz[2], and Juan Rodríguez-Hortalá[1]

[1] Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
fraguas@sip.ucm.es, juanrh@fdi.ucm.es
[2] INRIA Nancy & LORIA
Stephan.Merz@loria.fr

Abstract. Modern functional-logic programming languages like Toy or Curry feature non-strict non-deterministic functions that behave under call-time choice semantics. A standard formulation for this semantics is the *CRWL* logic, that specifies a proof calculus for computing the set of possible results for each expression. In this paper we present a formalization of that calculus in the Isabelle/HOL proof assistant. We have proved some basic properties of *CRWL*: closedness under c-substitutions, polarity and compositionality. We also discuss some insights that have been gained, such as the fact that left linearity of program rules is not needed for any of these results to hold.

1 Introduction

Fully formalizing the (meta)theory of a programming language can be beneficial for developing its foundations. There is an increasing number of researchers (see e.g. [2]) sharing the conviction that the combination *formalization+mechanized theorem proving* must (and will) play a prominent role in programming languages research and technology. In particular, formalizations help to clarify overlooked aspects, to discover pitfalls, and even to provide new insights; moreover, formalized metatheories lead to mechanized reasoning about programs, giving reliable support to tools like certifying compilers or certified program transformations.

In this paper we formalize the semantics of functional logic programming (FLP), a well established paradigm (see [9]) integrating features of logic and functional languages. In modern FLP languages such as Curry [10] or Toy [14] programs are constructor based rewrite systems that may be non-terminating and non-confluent. Semantically this leads to the presence of non-strict and non-deterministic functions. The semantics adopted for non-determinism is *call-time choice* [11, 8], informally meaning that in any reduction, all descendants of a given subexpression must share the same value. The semantic framework *CRWL*[3] was proposed in [7, 8] to accomodate this view of non-determinism, and

* This work has been partially supported by the Spanish projects TIN2005-09207-C03-03 (MERIT-FORMS-UCM), S-0505/TIC/0407 (PROMESAS-CAM) and TIN2008-06622-C03-01/TIN (FAST-STAMP).

[3] *CRWL* stands for "Constructor-based ReWriting Logic".

is nowadays considered the standard semantics of FLP. For the purpose of this paper, the most relevant aspect of *CRWL* is a proof calculus devised to prove reduction statements of the form $\mathcal{P} \vdash e \twoheadrightarrow t$, meaning that t is a possible (partial) value to which e can be reduced using the program $\mathcal{P}$.

We have chosen Isabelle/HOL as concrete logical framework for our formalization. Using such a broadly used system is not only easier, but also more flexible and stable than developing language specific tools like has been done, e.g., for logic programming [15] or functional programming [6].

The remainder of the paper is organized as follows: Sect. 2 contains some preliminaries about the *CRWL* framework , Sect. 3 presents the Isabelle theories developed to formalize *CRWL*, and Sect. 4 gives the mechanized proofs of some important properties of *CRWL*. Finally, Sect. 5 summarizes some conclusions and points to future work.

An extended version of this paper can be found at `http://gpd.sip.ucm.es/juanrh/pubs/isabell-crwl-report.pdf`. The Isabelle code underlying the results presented here is available at `https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/IsabelleCrwl`.

2 Preliminaries

2.1 Constructor-based term rewrite systems

We consider a first-order signature $\Sigma = CS \cup FS$, where CS and FS are two disjoint sets of *constructor* and defined *function* symbols respectively, each with associated arity. We write CS^n (FS^n resp.) for the set of constructor (function) symbols of arity n. The set *Exp* of *expressions* is inductively defined as

$$Exp \ni e ::= X \mid h(e_1, \ldots, e_n),$$

where $X \in \mathcal{V}$, $h \in CS^n \cup FS^n$ and $e_1, \ldots, e_n \in Exp$. The set *CTerm* of *constructed terms* (or *c-terms*) is defined like *Exp*, but with h restricted to CS^n (so $CTerm \subseteq Exp$). The intended meaning is that *Exp* stands for evaluable expressions, i.e., expressions that can contain function symbols, while *CTerm* stands for data terms representing values. We will write $e, e', \ldots$ for expressions and $t, s, \ldots$ for c-terms. The set of variables occurring in an expression e will be denoted as $var(e)$. We will frequently use *one-hole contexts*, defined as

$$Cntxt \ni \mathcal{C} ::= [\,] \mid h(e_1, \ldots, \mathcal{C}, \ldots, e_n)$$

for $h \in CS^n \cup FS^n$. The application of a context $\mathcal{C}$ to an expression e, written $\mathcal{C}[e]$, is defined inductively by

$$[\,][e] = e \quad \text{and} \quad h(e_1, \ldots, \mathcal{C}, \ldots, e_n)[e] = h(e_1, \ldots, \mathcal{C}[e], \ldots, e_n).$$

The set *Subst* of *substitutions* consists of finite mappings $\theta : \mathcal{V} \longrightarrow Exp$ (i.e., mappings such that $\theta(X) \neq X$ only for finitely many $X \in \mathcal{V}$), which extend naturally to $\theta : Exp \longrightarrow Exp$. We write $e\theta$ for the application of θ to e, and $\theta\theta'$

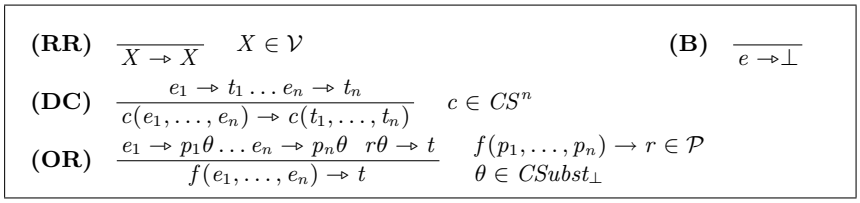

Fig. 1. Rules of *CRWL*

for the composition of substitutions, defined by $X(\theta\theta') = (X\theta)\theta'$. The domain of θ is defined as $dom(\theta) = \{X \in \mathcal{V} \mid X\theta \neq X\}$. In most cases we will use *c-substitutions* $\theta \in CSubst$, for which $X\theta \in CTerm$ for all $X \in dom(\theta)$.

A CRWL-*program* (or simply a *program*) is a set of rewrite rules of the form $f(\overline{t}) \to e$ where $f \in FS^n$, $e \in Exp$ and $\overline{t}$ is a linear n-tuple of c-terms, where linearity means that each variable occurs only once in $\overline{t}$. Notice that we allow e to contain *extra variables*, i.e., variables not occurring in $\overline{t}$. *CRWL*-programs often allow also conditions in the program rules. However, *CRWL*-programs with conditions can be transformed into equivalent programs without conditions, therefore we consider only unconditional rules.

2.2 The *CRWL* framework

In order to accomodate non-strictness at the semantic level, we enlarge Σ with a new constant constructor symbol $\bot$. The sets $Exp_\bot$, $CTerm_\bot$, $Subst_\bot$, $CSubst_\bot$ of partial expressions, etc., are defined naturally. Notice that $\bot$ does not appear in programs. Partial expressions are ordered by the *approximation* ordering $\sqsubseteq$ defined as the least partial ordering satisfying

$$\bot \sqsubseteq e \quad \text{and} \quad e \sqsubseteq e' \Rightarrow \mathcal{C}[e] \sqsubseteq \mathcal{C}[e'] \text{ for all } e, e' \in Exp_\bot, \mathcal{C} \in Cntxt$$

This partial ordering can be extended to substitutions: given $\theta, \sigma \in Subst_\bot$ we say $\theta \sqsubseteq \sigma$ if $X\theta \sqsubseteq X\sigma$ for all $X \in \mathcal{V}$.

The semantics of a program $\mathcal{P}$ is determined in *CRWL* by means of a proof calculus (see Fig. 1) for deriving reduction statements $\mathcal{P} \vdash e \twoheadrightarrow t$, with $e \in Exp_\bot$ and $t \in CTerm_\bot$, meaning informally that t is (or approximates) a *possible value* of e, obtained by iterated reduction of e using $\mathcal{P}$ under call-time choice. Rule B (bottom) allows us to avoid the evaluation of any expression, in order to get a non-strict semantics. Rules RR (restricted reflexivity) and DC (decomposition) allow us to reduce any variable to itself, and to decompose the evaluation of an expression whose root symbol is a constructor. Rule OR (outer reduction) expresses that to evaluate a function call we must first evaluate its arguments to get an instance of a program rule, perform parameter passing (by means of a $CSubst_\bot$ θ) and then reduce the instantiated right-hand side. The use of partial c-substitutions in OR is essential to express call-time choice, as only single partial values are used for parameter passing. Notice also that by the effect of θ in OR

extra variables in the right-hand side of a rule can be replaced by any c-term, but not by any expression. The *CRWL-denotation* of an expression $e \in Exp_\bot$ is defined as $[\![e]\!]^{\mathcal{P}} = \{t \in CTerm_\bot \mid \mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t\}$.

3 Formalizing *CRWL* in Isabelle

3.1 Basic definitions

We describe our formalization of *CRWL* in Isabelle. The first step is to define elementary types for the syntactic elements.

```
datatype signat = fs string | cs string
datatype varId = vi string
datatype exp = perp | Var varId | Ap signat "exp list"
types
  subst = "varId ⇒ exp option"
  rule = "exp * exp"
  program = "rule set"
```

Signatures are represented by a datatype that provides two constructors `cs` and `fs` to distinguish between constructor and function symbols. The type `varId` is used to represent variable identifiers, which will be employed to define substitutions. Then the datatype `exp` is naturally defined following the inductive scheme of $Exp_\bot$, therefore with this representation every expression is partial by default.

Substitutions (type `subst`) are represented as partial functions from variable identifiers to expressions, using Isabelle's `option` type. Hence the domain of a substitution ϑ will be the set of elements from `varId` for which ϑ returns some value different from `None`. Note that this representation does not ensure that domains of substitutions are finite. Our proofs do not rely on this finiteness assumption. Finally we represent a program rule as a pair of expressions, where the first element is considered the left-hand side of the rule and the second the right-hand side, and a program simply as a set of program rules. The set of valid *CRWL* programs is characterized by a predicate `crwlProgram :: "program ⇒ bool"` that checks whether the restrictions of left-linearity and constructor discipline are satisfied.

We define a function `apSubst :: "subst ⇒ exp ⇒ exp"` for applying a substitution to an expression. The composition of substitutions is defined through a function `substComp :: "subst ⇒ subst ⇒ subst"`. The following lemma ensures the correctness of this definition.

```
lemma subsCompAp :
  "(apSubst ϑ (apSubst σ e)) = (apSubst (substComp ϑ σ) e)"
```

Just as ML, the Isabelle type system does not support subtyping, which could have been useful to represent the sets of c-terms and c-substitutions. Instead, we define predicates `cterm` and `csubst` characterizing these subtypes. We prove the expected lemmas, such as that the composition of two c-substitutions is a c-substitution, or that the application of a c-substitution to a c-term yields a c-term.

3.2 Approximation order and contexts

Two key notions of *CRWL* have not yet been formalized: the approximation order $\sqsubseteq$, which will be used in the formulation of the polarity of *CRWL*, and the notion of one-hole context, which will be used in the compositionality.

The following inductively defined predicate `ordap` (with concrete infix syntax $\sqsubseteq$) models the approximation order.

```
inductive
  ordap :: "exp ⇒ exp ⇒ bool" ("_ ⊑ _" [51,51] 50)
where
  B: "perp ⊑ e"
| V: "Var x ⊑ Var x"
| Ap: "⟦ size es = size es' ; ALL i < size es. es!i ⊑ es'!i ⟧
       ⟹ Ap h es ⊑ Ap h es'"
```

Rule `B` asserts that `perp` $\sqsubseteq$ `e` holds for every `e`; rule `V` is needed for $\sqsubseteq$ to be reflexive; finally rule `Ap` ensures closedness under Σ-operations, and thus compatibility with context [3], because $\sqsubseteq$ is reflexive and transitive, as we will see. The following results state that our formulation of $\sqsubseteq$ defines a partial order.

```
lemma ordapRefl : "e ⊑ e"
lemma ordapTrans :
  assumes "e1 ⊑  e2" and "e2 ⊑ e3"
  shows "e1 ⊑  e3"
lemma ordapAntisym :
  assumes "e1 ⊑ e2" and "e2 ⊑ e1"
  shows "e1 = e2"
definition ordap_less ("_ ⊏ _" [51,51] 50) where
  "e ⊏ e' ≡ e ⊑ e' ∧ e ≠ e'"
interpretation exp : order [ordap ordap_less]
```

Contexts are represented as the datatype `cntxt`, defined as follows:

```
datatype cntxt = Hole | Cperp | CVar varId
               | CAp signat "cntxt list"
```

Note that `cntxt` cannot follow the inductive structure of *Cntxt* with precision, because the type system of Isabelle is not expressive enough to allow us to specify that only one of the arguments of `CAp` will be a context and the others will be expressions. Then our contexts are defined as expressions with possible some holes inside. Therefore the datatype `cntxt` represents contexts with any number of holes, even zero holes, and the function `apCon :: "exp ⇒ cntxt ⇒ exp"` is defined so it puts the argument expression in every hole of the argument context. In order to characterize contexts with just one hole, we define a function `numHoles :: "cntxt ⇒ nat"` that returns the numbers of holes in a context. Using it we can define define predicates `oneHole` and `noHole` and prove the following lemmas.

```
lemma noHoleApDontCare :
  assumes "noHole xC"
  shows "apCon e xC = apCon e' xC"

lemma oneHole :
  assumes "oneHole (CAp h xCs)"
  shows "∃ xC yCs zCs. xCs = (yCs @ xC # zCs) ∧ oneHole xC ∧
                       (∀c ∈ set (yCs @ zCs). noHole c)"
```

3.3 The *CRWL* logic in Isabelle/HOL

The *CRWL* logic has been formalized through the inductive predicate `clto` with infix notation `"_ ⊢ _ → _"`. The rules defining `clto` faithfully follow the inductive structure of the definition of *CRWL* as it is presented in Fig. 1.

```
inductive
  clto :: "program ⇒ exp ⇒ exp ⇒ bool"  ("_ ⊢ _ → _"
[100,50,50] 38)
where
  B[intro]:  "prog ⊢ exp → perp"
| RR[intro]: "prog ⊢ Var v → Var v"
| DC[intro]: "⟦size es = size ts;
               ∀i < size es. prog ⊢ es!i → ts!i
              ⟧ ⟹ prog ⊢ Ap (cs c) es → Ap (cs c) ts"
| OR[intro]: "⟦(Ap (fs f) ps, r) ∈ prog ; csubst ϑ ;
               size es = size ps ;
               ∀i < size es. prog ⊢ es!i → apSubst ϑ (ps!i);
               prog ⊢ apSubst ϑ r → t
              ⟧ ⟹ prog ⊢ Ap (fs f) es → t"
```

Using `clto` we can easily define the *CRWL* denotations in Isabelle as follows.

```
definition den :: "program ⇒ exp ⇒ exp set" where
  "den P e = {t. P ⊢ e → t}"
```

4 Reasoning about *CRWL* in Isabelle

The first interesting property that we are proving about *CRWL* expresses that evaluation is *closed under c-substitutions*: reductions are preserved when terms are instantiated by c-substitutions.

```
theorem crwlClosedCSubst :
  assumes "prog ⊢ e → t" and "csubst ϑ"
  shows "prog ⊢ apSubst ϑ e → apSubst ϑ t"
```

The proof of this lemma proceeds by induction on the *CRWL*-proof of the hypothesis, therefore we will have one case for each *CRWL* rule. The first three cases are proved automatically. However, to prove the case for rule `OR` Isabelle needs some help from us. We need to prove

```
prog ⊢ (Ap (fs f) (map (apSubst ϑ) es)) → (apSubst ϑ t)
```

and then let the simplifier apply the definition of `apSubst`. In the proof for that subgoal we used lemma `CSubsComp` to ensure that the c-substitution μ used for parameter passing composed with the c-substitution ϑ in the hypothesis yields another c-substitution, and lemma `subsCompAp` to guarantee the correct behaviour of the composition for those c-substitutions.

Note that for this result to hold no additional hypotheses about the program or the expressions involved are needed. In particular, this implies that the result holds even for programs that do not follow the constructor discipline or that have non left-linear rules. The Isabelle proof clearly shows that the important ingredients are the use of c-substitutions for parameter passing and the reflexivity of *CRWL* wrt. c-terms, expressed by lemma `ctermRefl`, which allows us to reduce to itself any expression `X`ϑ coming from a premise `X` $\rightarrow$ `X`.

The second property that we address is the *polarity of* CRWL. This property is formulated by means of the approximation order and roughly says that if we can compute a value for an expression then we can compute a smaller value for a bigger expression. Here we should understand the approximation order as an information order, in the sense that the bigger the expression, the more information it gives, and so more values can be computed from it.

```
theorem crwlPolarity :
  assumes "prog ⊢ e → t" and "e ⊑ e'" and "t' ⊑ t"
  shows "prog ⊢ e' → t'"
using assms proof (induct arbitrary: e' t')
```

The idea of the proof is to construct a *CRWL*-proof for the conclusion from the *CRWL*-proof of the hypothesis, hence it is natural to proceed by induction on the structure of this proof (method `induct`). The qualifier `arbitrary` is used to generalize the assertion for any expressions `e'` and `t'`. The proof also relies on the following additional lemmas about the approximation order, which were proved automatically by Isabelle.

```
lemma ordapPerp: assumes "e ⊑ perp" shows "e = perp"
lemma ordapVar: assumes "Var v ⊑ e" shows "e = Var v"
lemma ordapVar_converse:
   assumes "e ⊑ Var v" shows "e = perp ∨ e = Var v"
lemma ordapAp:
   assumes "Ap h es ⊑ e'"
   shows "∃es'. e' = Ap h es' ∧ size es = size es'
                ∧ (ALL i < size es. es!i ⊑ es'!i)"
lemma ordapAp_converse:
   assumes "e' ⊑ Ap h es"
   shows "e' = perp ∨
          (∃es'. e' = Ap h es' ∧ size es = size es'
                 ∧ (ALL i < size es. es'!i ⊑ es!i))"
```

The inductive proof for Thm. `crwlPolarity` again considers each *CRWL* rule in turn. In the case for `B` we have `t = perp`, hence we just have to apply `ordapPerp` to get `t' = perp`, and then use the *CRWL* rule `B`. Regarding `RR`, as then `t =`

`Var v`, by `ordapVar_converse` we get that either `t' = perp` or `t' = Var v`. The first case is trivial, and in the latter we just have to apply `ordapVar` getting `e' = Var v`, which is enough for Isabelle to finish the proof automatically. The case of `DC` is more complicated. Again we obtain two cases for `t' = perp` and `t'` a constructor application, by using lemma `ordapAp_converse`. While the first case is trivial, the second one requires some involved reasoning over the list of arguments, using the information we get from applying lemma `ordapAp`. Finally, the proof for `OR` is similar to the second case of the proof for `DC`, with a similar manipulation of the list of arguments, and the use of lemma `ordapAp` to obtain the induction hypothesis for the arguments.

Once again we find that this proof does not require any hypothesis on the linearity or the constructor discipline of the program: this is indeed quite obvious because this property only talks about what happens when we replace some subexpression by `perp`.

Finally we will tackle the *compositionality of* CRWL, that says that if we take a context with just one hole and an expression, then the set of values for the expression put it that context will be the union of the set of values for the result of putting each value for the expression in that context.

```
theorem compCRWL :
  assumes "oneHole xC"
  shows "den P (apCon e xC) =
         (⋃t∈den P e. den P (apCon t xC))"
```

We have proved the two set inclusions separately as auxiliary lemmas `compCRWL1` and `compCRWL2`. The proofs of these lemmas are quite laborious but essentially proceed by induction on the *CRWL*-proof in their hypothesis, using it to build a *CRWL*-proof for the statement in the conclusion. In these proofs, Lemma `noHoleApDontCare` from Subsect. 3.2 is fundamental.

Again, while Thm. `compCRWL` requires the context to have just one hole, it does not assume the linearity or constructor discipline of the program. This came as a surprise to us, and initially made us doubt about the accuracy of our formalization of *CRWL*. But it turns out that although *CRWL* is designed to work with *CRWL*-programs, that fulfil these restrictions, it can also be applied to general programs. For those programs some properties, such as the theorems `crwlClosedCSubst`, `crwlPolarity` and `compCRWL` still hold, but other fundamental properties do not, in particular the strong adequacy results w.r.t. its operational counterparts of [8, 12, 1]. The point it that for those programs *CRWL* does not deliver the "intended semantics" anymore. And this is not strange, because that semantics was intended with *CRWL*-programs in mind. For example, consider the non linear program $\mathcal{P} = \{f(X, X) \to a\}$. There is a *CRWL*-proof for the statement $\mathcal{P} \vdash f(a, b) \twoheadrightarrow a$ but this value cannot be computed in any of the operational notions of [8, 12, 1] nor in any implementation of FLP, in which the independancy of the matching process of the arguments — derived from left-linearity of program rules — is assumed. It is also not very natural that $f(a, b)$ could yield the value a for the arguments a and b being different values, which implies that the semantics defined by *CRWL* for non left-linear

programs is pretty odd. But that is not a big problem, because we only care about the properties of *CRWL* for the kind of programs it has been designed to work with. And if it enjoys some interesting properties for a bigger class of programs that is fine, because that nice properties will be inherited by the class of *CRWL*-programs.
On the other hand, for programs not following the constructor discipline, we will not even be able to have a matching for an argument of a rule which is not a constructor, because in the rule OR we have to reduce every argument of a function call to a value, which will be a c-term by Lemma `ctermVals` (see the extended version of this paper), and so could never be an instance of expression containing function symbols. Thus, the rule OR could not be used for program rules not following the constructor discipline.

5 Conclusions

This paper presented a formalization of the essentials of *CRWL* [7, 8], a well-known semantic framework for functional logic programming, in the interactive proof assistant Isabelle/HOL. We chose that particular logical framework for its stability and its extensive libraries. The Isar proof language allowed us to structure the proofs so that they become quite elegant and readable, as it is apparent looking at our Isabelle code.

Our formalization is generic with respect to syntax, and includes important auxiliary notions like substitutions or contexts. This is in contrast to previous work [4, 5] that focused on formalizing the semantics of each concrete program. In contrast, our paper focuses on developing the metatheory of the formalism, allowing us to obtain results that are more general and also more powerful: we formally prove essential properties of the paradigm like *polarity* or *compositionality* of the *CRWL*-semantics. We plan to extend our theories so that we will be able to reason about properties of concrete programs by deriving theorems that express verification conditions in the line of those stated in [4, 5].

While developing the formalization we realized an interesting fact not pointed out before: properties like polarity or compositionality do not depend on the constructor discipline and left-linearity imposed to programs. However, such requirements will certainly play an essential role when extending our work to formally relate the *CRWL*-semantics with operational semantics like the one developed in [12], one of our intended subjects of future work. We think that could be interesting in several ways. First of all it would be a further step in the direction of challenge 3 of [2], "Testing and Animating wrt the Semantics", because we would end up getting an interpreter of *CRWL* during the process. We should then also formalize the evaluation strategy for the operational semantics, obtaining an Isabelle proof of its optimality. Finally there are precedents [13, 12] of how the combination of a denotational and operational perspective is useful for general semantic reasoning in FLP.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.
2. B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In J. Hurd and T. F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005.
3. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, United Kingdom, 1998.
4. J. Cleva, J. Leach, and F. López-Fraguas. A logic programming approach to the verification of functional-logic programs. In *Proc. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming, PPDP'04*, pages 9–19. ACM, 2004.
5. J. Cleva and I. Pita. Verification of CRWL programs with rewriting logic. *J. Universal Computer Science*, 12(11):1594–1617, 2006.
6. M. de Mol, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Theorem proving for functional programmers. In T. Arts and M. Mohnen, editors, *Implementation of Functional Languages, 13th International Workshop, IFL 2002*, volume 2312 of *Lecture Notes in Computer Science*, pages 55–71. Springer, 2001.
7. J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In *Proc. European Symposium on Programming (ESOP'96)*, pages 156–172. Springer LNCS 1058, 1996.
8. J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47–87, 1999.
9. M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.
10. M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). Available at *http://www.informatik.uni-kiel.de/~curry/report.html*, March 2006.
11. H. Hussmann. *Non-Determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag, 1993.
12. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In *Proc. Principles and Practice of Declarative Programming*, pages 197–208. ACM Press, 2007.
13. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Rewriting and call-time choice: the HO case. In *Proc. 9th International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *LNCS*, pages 147–162. Springer, 2008.
14. F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
15. R. F. Stärk. The theoretical foundations of lptp (a logic program theorem prover). *J. Log. Program.*, 36(3):241–269, 1998.

SAT Solver Verification Project*

Filip Marić and Predrag Janičić

Faculty of Mathematics, University of Belgrade,
Belgrade, Studentski Trg 16, Serbia
{filip, janicic}@matf.bg.ac.rs

Abstract. In this paper we give an overview of our SAT solver verification project. This is the first paper to present this project as a whole. We summarize the results achieved in the verification of SAT solvers described in terms of abstract state transition systems, in the Hoare-style verification of an imperative implementation of a modern SAT solver, and in generation of a trusted SAT solver based on the shallow embedding into HOL. Our formalization and verification are accompanied by a solver implemented in C++ and a trusted, automatically generated solver implemented in a functional language. One of the main final goals of our project is reaching to a both efficient and fully trusted SAT solver. Other goals include rigorous analyzes of existing SAT solving systems.

1 Introduction

One of the most important goals of computer science is reaching trusted software. This is especially important for algorithms and programs that have numerous applications, including SAT solvers — programs that test satisfiability of propositional formulae (usually given in conjunctive normal form). The SAT problem is the first problem that was proved to be NP-complete [Coo71] and it still holds a central position in the field of computational complexity. The majority of the state-of-the-art complete SAT solvers are based on the backtracking algorithm called Davis-Putnam-Logemann-Loveland (DPLL) [DP60,DLL62]. Spectacular improvements in the performance of SAT solvers have been achieved in the last decade and nowadays SAT solvers can decide satisfiability of propositional formulae with tens of thousands of variables and millions of clauses. Thanks to these advances, modern SAT solvers can handle more and more practical problems in areas such as electronic design automation, software and hardware verification, artificial intelligence, operations research.

The tremendous advance in the SAT solving technology has not been accompanied with corresponding theoretical results about solvers' correctness. Descriptions of new algorithms and techniques are usually given in terms of implementations, while correctness arguments are either not given or are given only in outlines. This gap between practical and theoretical progress needs to be filled and first steps in that direction have been made only recently.

* This work was partially supported by Serbian Ministry of Science grant 144030.

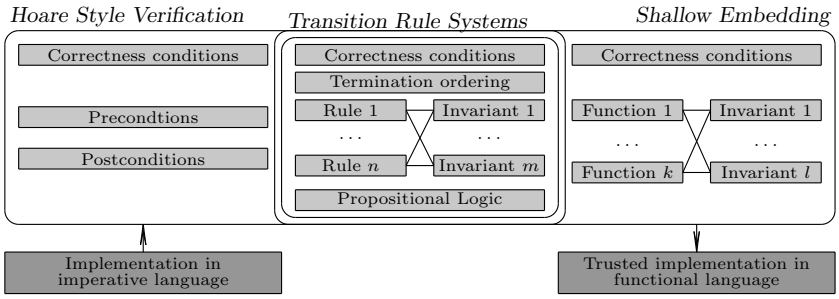

Fig. 1. Overall structure of the SAT solver verification project

One approach for achieving a higher level of confidence in SAT solvers' results, successfully used in recent years, is *proof-checking*. Solvers are modified so that they output not only *SAT* or *UNSAT* answers, but also evidences for their claims (models for satisfiable and proof-objects for unsatisfiable instances) which are then checked by independent checkers. Proof-checking is relatively easy to implement, but it has some drawbacks. Generating proof-objects introduces some runtime and storage overheads (proofs are typically large and may consume gigabytes of storage space) [Gel07]. Since proof-checkers have to be trusted, they must be very simple programs so they can be ,,verified" by code inspection.

Another approach is to verify a SAT solver itself, instead of checking each of the solver's claims. This approach is much harder to realize (since it requires formal analysis of the complete solver's behaviour) but is much more rewarding:

- Although the overheads of generating unsatisfiability proofs during solving are not unmanageable, they can still be avoided if the solver itself is trusted.
- Verification of modern SAT solvers could help in better theoretical understanding of how and why they work and their rigorous analysis may reveal some possible improvements in underlying algorithms and techniques.
- Verified SAT solvers can serve as trusted kernel checkers for verifying results of other untrusted verifiers (e.g., BDDs, model checkers, SMT solvers) [SV09]. Also, verification of some SAT solver modules (e.g., BCP) can serve as a basis for creating both verified and an efficient proof-checkers for SAT.
- In addition to the above benefits, we want to demonstrate that, thanks to the recent advances in software verification technology, the time has finally come when it is possible to have a non-trivial, widely used software fully verified. Such work would contribute to the *Verification Grand Challenge*.

In this paper we present our ongoing project on SAT solver verification, with largest parts already completed. The project aims at producing solvers that are both efficient and fully trusted. In order to achieve the desired, highest level of trust, a fully mechanized and machine-checkable formalization is being developed. An overall structure of our project is illustrated in Fig. 1. Within the

project, we consider three ways of specifying modern SAT solvers and the corresponding verification paradigms (each with its advantages and disadvantages):

Abstract state transition systems. We have formally verified several abstract state transition systems that describe SAT solvers [NOT06,KG07]. Verification of such systems proves to be of vital importance because it serves as a key building block in other approaches to formalization.

Imperative implementation. We have made a more detailed (compared to the abstract state transition systems) description of a SAT solver in an imperative pseudo programming language. In parallel, we have developed a corresponding SAT solver ArgoSAT in C++. Solver's properties have been formalized and verified within the Hoare logic.

Shallow embedding into a proof assistant. We have defined a SAT solver as a set of recursive functions within higher order logic of the system Isabelle (regarded as a pure functional language) and its correctness has been formally proved. Based on this specification, an executable functional program has been generated by means of the *code extraction*.

In the rest of this paper, due to the lack of space, we give just a very few used definitions and just briefly comment only the central theorems and proofs. All the definitions, conjectures, and proofs have been completely formalized and verified within the Isabelle/Isar system [NPW02] and the complete proof documents are available in [Mar08]. Parts of the described project have been already described elsewhere [MJ09a,Mar09a,Mar09b,Mar09c,MJ09b], but in this paper the project is described as a whole for the first time. For a detailed survey of modern SAT solving technology and algorithms we refer the interested reader to other sources on these matters (e.g, [BHM+09]).

2 Formalization of Logic of Propositional CNF Formulae

The syntax of CNF formulae is based on the following types.

Definition 1. *A* Variable *is identified with a natural number. A* Literal *is either a positive variable (*Pos *vbl) or a negative variable (*Neg *vbl). A* Clause *is a list of literals. A* CNF Formula *is a list of clauses.*

Several basic operations on these types are introduced (e.g., *variable of a literal* l, denoted by (var l), the set of all variables that occur in a formula F, denoted by (vars F), *the opposite literal of a literal* l, denoted by $\bar{l}$).

The semantics of CNF formulae is based on the notion of *valuation*.

Definition 2. *A* Valuation *is a list of literals. For a given valuation* v*, a* literal l is true *(denoted* $v \vDash l$*) iff* $l \in v$*, and is* false *(denoted* $v \vDash \neg l$*) iff* $\bar{l} \in v$*. A* clause c is true*, denoted* $v \vDash c$*, iff* $\exists l.\ l \in c \wedge v \vDash l$*, and is* false *(denoted* $v \vDash \neg c$*) iff* $\forall l.\ l \in c \Rightarrow v \vDash \neg l$*. A* formula F is true *(denoted* $v \vDash F$*) iff* $\forall c.\ c \in F \Rightarrow v \vDash c$*, and is* false *(denoted* $v \vDash \neg F$*) iff* $\exists c.\ c \in F \wedge v \vDash \neg c$*. A valuation* v *is* consistent*, denoted* (consistent v)*, iff it does not contain both a literal and its opposite. A* model *of a formula* F *is a consistent valuation in which* F *is true. A formula* F *is* satisfiable*, denoted* (sat F)*, iff it has a model.*

Note that, although total valuations are usually defined as Boolean variable assignments, the given definition that covers also partial valuations and more closely relates to the internal working of modern SAT solvers. The confidence in our correctness proofs for a SAT solver, in bottom line, relies on the given definitions. Fortunately, they are rather simple and can be checked by human inspection. Still, in order to prove correctness conditions, many additional notions have to be introduced and their properties have to be formally proved (e.g., entailment of a literal or a clause by a formula, denoted by $F \models l$ or $F \models c$, logical equivalence of two formulae, denoted by $F_1 \equiv F_2$).

SAT solving related notions. Some notions specific to SAT solving are also introduced within our formalization. For example, a unit clause c (denoted by $\mathsf{isUnit}\ c\ l\ v$), is a clause which contains a literal l undefined in v and whose all other literals are false in v; a reason clause c for the literal l (denoted by $\mathsf{isReason}\ c\ l\ v$), is a clause that contains l (true in v), whose all other literals are false in v, and their opposites precede l in v; the resolvent of two clauses (denoted by $\mathsf{resolve}\ c_1\ c_2\ l$), etc.

Modern SAT solvers slightly extend the notion of valuation by distinguishing two different kinds of literals: *decision* and *implied*. For example, a trail M could be $[+1, |-2, +6, |+5, -3, |-7]$. Decision literals are marked by the symbol | and they split the trail into levels, so M has 4 different levels (labelled by 0 to 3): $+1$, then $-2, +6$, then $+5, -3$, and -7. There is a number of operations on assertion trails used within SAT solvers. These operations have also been formally defined within our theory and their properties have been formally proved. Some of these are the list of decisions in a trail (denoted by $(\mathsf{decisions}\ M)$), the list of decisions that precede the first occurrence of a given literal (denoted by $\mathsf{decisionsTo}\ l\ M$)), the number of levels in a trail (denoted by $(\mathsf{currentLevel}\ M)$), prefix of a trail up to the given level (denoted by $(\mathsf{prefixToLevel}\ level\ M)$), etc.

3 Verification of the State Transition Systems

Modern DPLL-based SAT solvers can be modelled as abstract state transition systems (ASTS). Such systems define the top-level architecture of SAT solvers as mathematical objects that can be rigorously reasoned about and whose correctness is expressed in pure mathematical terms. During the last few years two such systems have been proposed [NOT06,KG07], both accompanied by informal, pen-and-paper correctness proofs. We used ASTS from [KG07] as a starting point and developed a slightly modified ASTS shown in Fig. 2. The system models the solver's behaviour as transitions between states that represent values of the global variables of the solver. Transitions are performed only by using the transition rules. The rules have guarded assignment form: above the line is a condition that enables the rule application, below the line is an update to the state variables. The solving process is finished when no transition rule applies.

Our system has been formalized in the following way. A *state* $(F, M, C, conflict)$ consists of a formula F being tested for satisfiability, a trail M, a conflict analysis clause C, and a Boolean variable $conflict$ that flags if the current for-

Decide:
$$\frac{l \in F_0 \qquad l, \bar{l} \notin M}{M := M\ |l}$$

UnitPropagate:
$$\frac{c \in F \qquad \mathsf{isUnit}\ c\ l\ M}{M := M\ l}$$

Conflict:
$$\frac{conflict = \bot \qquad c \in F \qquad M \vDash \neg c}{conflict = \top \qquad C := c}$$

Explain:
$$\frac{conflict = \top \quad l \in C \quad c \in F \quad \mathsf{isReason}\ c\ \bar{l}\ M}{C := \mathsf{resolve}\ C\ c\ l}$$

Backjump:
$$\frac{conflict = \top \quad C \in F \quad C = l \vee l_1 \vee \ldots \vee l_k \quad \mathsf{level}\ \bar{l} > m \geq \mathsf{level}\ \overline{l_i}}{conflict = \bot \qquad M := (\mathsf{prefixToLevel}\ m\ M)\ l}$$

Learn:
$$\frac{C \notin F}{F := F \cup C}$$

Forget:
$$\frac{conflict = \bot \quad c \in F \quad F \setminus c \vDash c}{F := F \setminus c}$$

Restart:
$$\frac{conflict = \bot}{M := \mathsf{prefixToLevel}\ 0\ M}$$

Fig. 2. Abstract state transition system for a DPLL-based SAT solver

mula is false in the current valuation (i.e., if the *conflict analysis* is under way). The rules have been formalized using relations over states. For instance,

$$\mathsf{unitPropagate}\ (M_1, F_1, C_1, conflict_1)\ (M_2, F_2, C_2, conflict_2) \iff$$
$$\exists c\ l.\ c \in F_1\ \wedge\ \mathsf{isUnit}\ c\ l\ M_1\ \wedge$$
$$M_2 = M_1 @ l\ \wedge\ F_2 = F_1\ \wedge\ C_2 = C_1\ \wedge\ conflict_1 = conflict_2$$

Two states are in relation $\rightarrow$ iff they are in one of the relations describing the transition rules. A state $([\,], F_0, [\,], \bot)$ is an *initial state* for the input formula F_0. A state s is *final state* with respect to $\rightarrow$, iff it is its minimal element, i.e., if there is no state s' such that $s \rightarrow s'$. A final state is an *accepting state* if it holds that $conflict = \bot$. A final state is a *rejecting state* if it holds that $conflict = \top$.

Theorem 1 (Correctness). *For any satisfiable input formula, the system consisting of the given rules terminates in an accepting state, and for any unsatisfiable formula, it terminates in an rejecting state.*

Our correctness proof for the above system is based on formulating a set of suitable invariants and a well-founded ordering defined on states that ensures termination (as illustrated in Fig. 1). For example, we proved that the following invariants hold for each state reached from an initial state.

$Invariant_M$: $\mathsf{consistent}\ M\ \wedge\ \mathsf{distinct}\ M$

$Invariant_{vars}$: $\mathsf{vars}\ M\ \cup\ \mathsf{vars}\ F \subseteq\ \mathsf{vars}\ F_0$

$Invariant_{equiv}$: $F \equiv F_0$

$Invariant_{impliedLiterals}$: $\forall l.\ l \in M\ \Longrightarrow\ F\ @\ (\mathsf{decisionsTo}\ l\ M) \vDash l$

$Invariant_C$: $conflict\ \Longrightarrow\ M \vDash \neg C\ \wedge\ F \vDash C$

$Invariant_{reasonClauses}$: $\forall\ l.\ l \in M\ \wedge\ l \notin (\mathsf{decisions}\ M)\ \Longrightarrow$
$\exists\ c.\ (\mathsf{isReason}\ c\ l\ M)\ \wedge\ F \vDash c$

The main advantage of the ASTSs is that they are mathematical objects, so it is relatively easy to make their formalization within higher order logic and to formally reason about them. Also, their verification can be a key building block for other verification approaches. Disadvantages are that the transition systems do not specify many details present in modern solvers' implementations and that they are not directly executable. More details on the verification of the ASTSs for SAT are given in [MJ09b].

4 Hoare-style Verification

Verification of imperative programs is usually done in the Floyd-Hoare logic [Hoa69]. Its central object is a *Hoare triple* of the form $\{P\}$ `code` $\{Q\}$. Hoare triple should be read as: "given that the *precondition* P holds before `code` is executed and the `code` execution terminates, the *postcondition* Q will hold at the point after `code` was executed". Hoare triples are manipulated by the inference rules that are formulated for each construct of the programming language.

Using this approach, we have verified the core of our solver ArgoSAT[1] implemented in C++[2]. Its implementation [Mar09b] closely follows the abstract state transition system given in Sect. 3, but also supports all standard techniques present in a modern SAT solver (e.g., MiniSAT [ES04]) not covered by the ASTS. Most important of these are the *two-watched literal unit propagation scheme* used for efficient detection of false and unit clauses in F wrt. the current trail M, *special treatment of single-literal clauses* which are directly asserted to the decision level zero of the trail M instead of adding them to F, and *efficient implementation of conflict analysis* using specialized data-structures for storing the conflict clause C.

Using the Hoare logic for the language complex as C++ was out of our reach. Therefore, we designed a pseudo language rich enough to support the implementation of our SAT solver, but simple enough to formulate a convenient Hoare logic axioms for all its constructs. The whole of the solver's core has been expressed within this pseudo language[3]. As an example, we list one function:

```
function applyUnitPropagate() : Boolean
begin
  assertLiteral ((head Q), false);
  Q := (tail Q);
end
```

Following the two-watched literal scheme, all unit literals are placed in a unit propagation queue Q from where they are taken and asserted to M. Therefore, a precondition for the applyUnitPropagate function is that all literals of Q are unit literals. The following example Hoare triple states that this property is preserved after the function call:

$$\{\forall l.\ l \in Q \longrightarrow \exists\ c.\ c \in F\ \wedge\ \mathsf{isUnit}\ c\ l\ M\}$$
$$\mathsf{applyUnitPropagate}()$$
$$\{\forall l.\ l \in Q \longrightarrow \exists\ c.\ c \in F\ \wedge\ \mathsf{isUnit}\ c\ l\ M\}$$

Heuristic components were specified only by Hoare triples. This way, for any implementation of a heuristic it suffices to prove that it meets the corresponding triple. For example, the selection of a literal for the Decide rule is specified as:

$\{\mathsf{vars}\ M \neq \mathsf{vars}\ F_0\}$ selectLiteral() $\{\mathsf{var}\ ret \in \mathsf{vars}\ F_0 \wedge \mathsf{var}\ ret \notin \mathsf{vars}\ M\}$

[1] The web page of ArgoSAT is `http://argo.matf.bg.ac.rs`.

[2] The core of ArgoSAT, implementing the rules given in Fig. 2 in an efficient way, counts around 1500 loc, while the whole system counts around 5000 loc.

[3] The description of the solver in the pseudo language is somewhat shorter then in C++, because of the simplified syntax.

Once the solver has been described in the pseudo programming language, the preconditions and postconditions for each fragment of the code are manually specified and joint together, following a suitable Hoare logic for our pseudo language. The entry point to the solver is the solve function which, if terminates, sets the value of $satFlag$ (either to SAT or $UNSAT$).

Theorem 2 (Partial correctness). *The SAT solver satisfies the Hoare triple:* $\{\top\}\ \mathsf{solve}(F_0)\ \{(satFlag = UNSAT \wedge \neg \mathsf{sat}\ F_0) \vee (satFlag = SAT \wedge M \vDash F_0)\}$

The main benefit of using the Hoare style verification is that it enabled us to address imperative code which is the way that most real-world SAT solvers are implemented. Thanks to this, the confidence in our solver ArgoSAT is higher compared to other C/C++ implementations. On the other hand, there is still a gap between our correctness proof and the C++ implementation. First, there is no formal link between C++ and our pseudo language implementation. Second, there has been a number of manual steps in formulating correctness conditions and joining them together. More details on our description of a solver in an imperative language and its Hoare-style verification are given in [Mar09a].

5 Shallow Embedding into HOL

When using the *shallow embedding into HOL* approach for verification, a program (a SAT solver in our case) is expressed as a set of recursive functions in HOL (for this purpose, treated as a pure functional programming language) and its properties are proved mainly by induction and equational reasoning.

Although a programming paradigm had to be changed from imperative to pure functional, our implementation closely follows the one described in Sect. 4 and that is the core of our solver ArgoSAT. All aspects of the implementation that are present in the imperative implementation verified by the Hoare-style approach are also present in our functional implementation within Isabelle[4].

In an imperative or object-oriented language, the state of the solver is represented by using global or class variables. The solver functions access and change the state variables as their side-effects. In HOL, functions cannot have side-effects, so the solver state must be wrapped up in a record and passed around with each function call. For example, the following Isabelle record directly correspond to the state of the abstract state transition systems described in Sect. 3:

```
record State =
  "getF" :: Formula
  "getM" :: LiteralTrail
  "getC" :: Clause
  "getConflictFlag" :: Boolean
```

However, in order to have more advanced techniques implemented, the state had to be extended, and in our final definition it contains 14 components.

All functions in our functional implementation receive the current solver state as their parameter and return the modified state as their result. This explicit

[4] Formal definitions of the solver functions count over 500 lines of Isabelle code.

state passing can be hidden if standard *monadic combinators* are used. This support has been recently added to Isabelle along with a convenient Haskell-like do-syntax [BKH+08]. In this syntax, the applyUnitPropagate function becomes:

```
definition applyUnitPropagate :: "State ⇒ State"
where
"applyUnitPropagate =
do
  Q ← readQ; assertLiteral (hd Q) False;
  Q' ← readQ; updateQ (tl Q')
done"
```

Functions `readQ` and `updateQ` modify the Q component of the current state.

Once the solver has been defined in HOL, its properties are formally proved. The main result is the following correctness theorem.

Theorem 3 (Correctness). solve $F_0 =$ sat F_0

Again, it has been proved that all states that are reached during the code execution (this time these are the states that are returned by the functions of the solver) satisfy a given set of invariants (as illustrated in Fig. 1). These invariants include all invariants formulated for the abstract state transition systems, but also include additional ones (24 invariants in total). Therefore, it had to be proved that the code preserves all the additional invariants and it turned out that this task was equally hard (if not harder) as proving the properties of the ASTS. For termination, it was required to prove that the function `solve` is total. Only three functions called by it have been defined by general recursion and their termination is not trivial. Since the function solve is the only entry point to our solver, it was sufficient to prove termination of these functions only for those values of their input parameters that could actually be passed to them during a solver's execution starting from an initial state. We have used Isabelle's built-in features to model this kind of partiality [Kra08] and reused the orderings defined for abstract state transition systems to prove termination.

Unlike the Hoare-style approach that starts with an existing solver implementation, when using the shallow embedding approach, the executable code in one of the leading functional languages (Haskell, SML, or OCaml) can be exported by using the *code extraction*, supported by Isabelle.

Advantages of using the shallow embedding are that, once the solver is defined within the proof assistant, it is possible to perform its verification directly inside the logic and a formal model of the operational or denotational semantics of the language is not required. Also, executable code can be extracted and it can be trusted with a very high level of confidence. On the other hand, it is required to build a fresh implementation of a SAT solver within the logic. Also, special techniques must be used to have mutable data-structures and consequently, an efficient generated code. More details on the verification by shallow embedding are given in [Mar09c]. We used this approach also for verification of the classic DPLL procedure, and details are given in [MJ09a].

6 Related Work

First steps towards verification of SAT solvers have been made only recently. The authors of two transition rule systems for SAT informally proved their correctness [NOT06,KG07]. Zhang and Malik have informally proved correctness of a modern SAT solver [ZM03]. Lescuyer and Conchon have formalized, within the system Coq, a SAT solver based on the classic DPLL procedure [LS08]. Shankar and Vaucher have formally and mechanically verified a high level description of a modern DPLL-based SAT solver within the system PVS [SV09]. Although these approaches include most state-of-the art SAT algorithms, lower-level implementation techniques (e.g., two-watch unit propagation scheme) are not covered by any of these descriptions. Our project provides fully mechanized correctness proofs for modern SAT solvers within three verification paradigms with both higher and lower level state-of-the-art SAT techniques, and, as we are aware of, it is the only such formalization.

7 Conclusions and Future Work

In this paper we gave an overview of our ongoing project on the modern SAT solver verification. SAT solvers have been formalized in three different ways: as abstract state transition systems, as imperative pseudo programming language code, and as a set of recursive HOL functions. All three formalizations have been verified using appropriate paradigms. Each of them has its own advantages and disadvantages, making them in some aspects complementary and in some aspects overlapping. The complete formalization has been made within Isabelle/Isar proof assistant and is publicly available[5]. Although it is hard to quantify the efforts invested in formally proving correctness conditions described in this work, we estimate that we have, so far, invested around 1.5 man-years into this project. Although there are other attempts at proving correctness of modern SAT solvers, to our best knowledge, our project gives the most detailed formalized and fully verified descriptions of a modern SAT solver so far.

One of the main remaining tasks in our project is to increase the efficiency of the code exported from the shallow embedding specification. Implementation of some heuristic components has to be more involved. For example, currently we have implemented only a trivial decision heuristic that picks a random undefined literal, but in order to have a practically usable solver, an advanced decision heuristic (e.g., VSIDS) should be used. Also, several low-level algorithmic improvements have to be made. Although these modifications require more work, we believe that they are rather straightforward. However, the most problematic issue is the fact that because of the pure functional nature of HOL no side-effects are possible and there can be no *destructive updates* of data-structures. To overcome this problem, we are planning to instruct the code generator to generate monadic Haskell and imperative ML code which would lead to huge efficiency benefits since it allows mutable references and arrays [BKH$^+$08]. We hope that with these modifications, the generated code could become practically usable

[5] The proof scripts make around 30000 lines of Isabelle code.

and comparable to state-of-the-art SAT solvers and this is the subject of our current work.

References

[BHM+09] A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability*. IOS Press, 2009.

[BKH+08] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkok, and J. Matthews. Imperative functional programming with Isabelle/HOL. In *TPHOLs '08*, LNCS 5170, Montreal, 2008.

[Coo71] S. A. Cook. The Complexity of Theorem-Proving Procedures. In *3rd STOC*, New York, 1971.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-proving. *Commun. ACM* 5(7), pp. 394–397, 1962.

[DP60] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *J. ACM* 7(3), pp. 201–215, 1960.

[ES04] N. Een and N. Sorensson. An Extensible SAT Solver. In *SAT '03*, LNCS 2919, S. Margherita Ligure, 2003.

[Gel07] A. Van Gelder. Verifying Propositional Unsatisfiability: Pitfalls to Avoid. In *SAT '07*, LNCS 4501, Lisbon, 2007.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12(10), pp. 576–580, 1969.

[Kra08] A. Krauss. Defining recursive functions in Isabelle/HOL. `http://isabelle.in.tum.de/documentation.html`, 2008.

[KG07] S. Krstić and A. Goel. Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL. In *FroCos '07*, LNCS 4720, Liverpool, 2007.

[LS08] S. Lescuyer and S. Conchon A Reflexive Formalization of a SAT Solver in Coq. In *TPHOLs'08: Emerging Trends*, Montreal, 2008.

[Mar08] F. Marić, SAT Solver Verification. *The Archive of Formal Proofs,* `http://afp.sf.net/entries/SATSolverVerification.shtml`.

[Mar09a] F. Marić. Formalization and Implementation of SAT solvers. *J. Autom. Reason.* To appear. 2009.

[Mar09b] F. Marić. Flexible Implementation of SAT solvers. In preparation.

[Mar09c] F. Marić. Formal Verification of a Modern SAT Solver. Manuscript submitted.

[MJ09a] F. Marić and P. Janičić. Formal Correctness Proof for DPLL Procedure. *Informatica*. To appear. 2009.

[MJ09b] F. Marić, P. Janičić. Formalization of Abstract State Transition Systems for SAT. In preparation.

[NOT06] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *J. of the ACM* 53(6), pp. 937–977, 2006.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, LNCS 2283, Springer, 2002.

[SV09] N. Shankar and M. Vaucher. The mechanical verification of a DPLL-based satisfiability solver. In preparation.

[ZM03] L. Zhang and S. Malik. Validating SAT Solvers Using Independent Resolution-Based Checker. In *DATE '03*, Münich, 2003.

A Theorem Proving Approach Towards Declarative Networking

Anduo Wang[1] Boon Thau Loo[1]
Changbin Liu[1] Oleg Sokolsky[1] Prithwish Basu[2]

[1] Computer and Information Sciences Department, University of Pennsylvania,
3330 Walnut Street, Philadelphia, PA 19104-6389
[2] Network Research Group, BBN Technologies,
10 Moulton Street, Cambridge, MA 02138
{anduo,changbl,boonloo,sokolsky}@seas.upenn.edu pbasu@bbn.com

Abstract. We present the *DRIVER* system for designing, analyzing and implementing network protocols. *DRIVER* leverages declarative networking, a recent innovation that enables network protocols to be concisely specified and implemented using declarative languages. *DRIVER* takes as input declarative networking specifications written in the Network Datalog (*NDlog*) query language, and maps that automatically into logical specifications that can be directly used in existing theorem provers to validate protocol correctness. As an alternative approach, network designer can supply a component-based model of their routing design, automatically generate PVS specifications for verification and subsequent compilation into verified declarative network implementations. We demonstrate the use of *DRIVER* for synthesizing and verifying a variety of well-known network routing protocols.

1 Introduction

In this paper, we present the *DRIVER* (*Declarative Routing Implementation and VERification*) system for designing, analyzing and implementing network protocols within a unified framework. Our work is a significant step towards *bridging* network specifications, protocol verification, and implementation within a common language and system. The *DRIVER* framework achieves this unified capability via the use of *declarative networking* [13, 12], a declarative domain-specific approach for specifying and implementing network protocols, and theorem proving, a well established verification technique based on logical reasoning.

DRIVER leverages our prior work on a *declarative network verifier (DNV)* [18] which demonstrates that one can leverages declarative networking's connection to logic programming by automatically compiling high-level declarative networking program written in the Network Datalog (*NDlog*) query language into formal specifications, which can be directly used in a theorem prover for verification. The proving process guided by the user is then carried out in a general-purpose theorem prover and proofs are mechanically checked. Declarative networking programs that have been verified in *DRIVER* can be directly executed as implementations, hence bridging specifications and implementations within a unified declarative framework.

In addition to verifying declarative networking programs using a theorem prover, the *DRIVER* system enables a similar transformation of verified formal specifications (limited to fragment of second order logic) to *NDlog* program

for execution. This enables a network designer to either directly verify network implementation specified in *NDlog* or conceptualize and verify the design of a network in components aided by a theorem prover prior to implementation.

2 Background

Declarative networks are implemented using *Network Datalog* (*NDlog*), a distributed logic-based recursive query language first introduced in the database community for querying network graphs. In prior work, it has been shown that traditional routing protocols can be implemented in a few lines of declarative code [13], and complex protocols in orders of magnitude less code [12] compared to traditional imperative implementations. This compact and high-level language enables rapid prototype development, ease of customization, optimizability, and the potentiality for protocol verification. When executed, these declarative networks perform efficiently relative to imperative implementations, as demonstrated by the *P2* declarative networking system [1].

2.1 Datalog Language

NDlog is primarily a distributed variant of Datalog. We illustrate *NDlog* using a simple example of two rules that computes all pairs of reachable nodes:

```
r1 reachable(@S,N) :- link(@S,N).
r2 reachable(@S,D) :- link(@S,N), reachable(@N,D).
Query reachable(@S,D).
```

The rules `r1` and `r2` specify a distributed transitive closure computation, where rule `r1` computes all pairs of nodes reachable within a single hop from all input links (denoted by `neighbor`), and rule `r2` expresses that "if there is a link from `S` to `N`, and `N` can reach `D`, then `S` can reach `D`."

NDlog supports a *location specifier* in each predicate, expressed with the `@` symbol followed by an attribute. This attribute is used to denote the source location of each corresponding tuple. For example, all `reachable` and `link` tuples are stored based on the `@S` address field. The output of interest is the set of all `reachable(@S,D)` tuples, representing reachable pairs of nodes from `S` to `D`.

2.2 Soft-state Storage Model

Declarative networking incorporates support *soft-state* [15] derivations commonly used in networks. In the soft state storage model, all data (input and derivations) has an explicit "time to live" (TTL) or lifetime, and all tuples must be explicitly reinserted with their latest values and a new TTL, or they are deleted.

The soft-state storage semantics are as follows. When a tuple is derived, if there exists another tuple with the same primary key but differs on other attributes, an *update* occurs, in which the new tuple replaces the previous one. On the other hand, if the two tuples are identical, a *refresh* occurs, in which the existing tuple is extended by its TTL.

For a given predicate, in the absence of any `materialize` declaration, it is treated as an *event* predicate with zero lifetime. Since events are not stored, they are primarily used to trigger rules periodically or in response to network events.

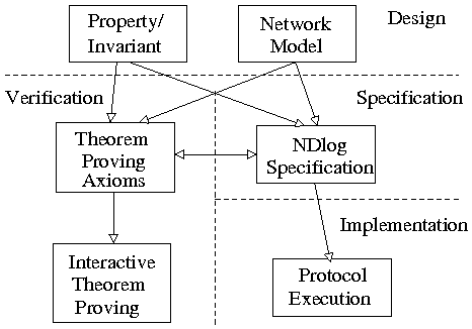

Fig. 1. Overview of DRIVER

3 Overview of DRIVER

Figure 1 provides an overview of *DRIVER*'s basic approach towards unifying specifications, verification, and implementation within a common declarative framework. The approach is broken up into the following four phases: *design*, *specification*, *verification*, and *implementation*.

In the initial design phase of *DRIVER*, a network designer develops a conceptual model for the routing protocol. In practice, this step may be optional, but having such a model is often useful both from the implementation standpoint, and for verifying one's protocol design.

Based on the design, two options are available. First, *NDlog* networking programs can be synthesized from the design, and then the *NDlog* implementations can be directly verified in an theorem prover. Second, the designer can first verify the design using a theorem prover and then automatically generate the corresponding *NDlog* program.

Considering the first option, *DRIVER* takes as input *NDlog* program representation of the routing protocol we are interested in. In order to carry out the formal verification process, the *NDlog* programs are automatically compiled into formal specifications recognizable by a standard theorem prover (e.g. PVS [2], Coq [3]) using the *axiom generator*, as depicted in the left-part of Figure 1.

At the same time, the protocol designer specifies high-level invariant properties of the protocol to be checked via two mechanisms: invariants can be written directly as theorems in the theorem prover, or expressed as *NDlog* rules which can be automatically translated into theorems using the axiom generator. The first approach increases the expressiveness of invariant properties, where one can reason with invariants that can be only expressible in higher order logic. The second approach has restricted expressiveness based on *NDlog*'s use of Datalog, but has the added advantage that the same properties expressed in *NDlog* can be verified in both theorem prover and checked at runtime.

From the perspective of network designers, as depicted in the left part of Figure 1, they reason about their protocols using the high-level protocol specifications and invariant properties.The *NDlog* high-level specifications are directly executed and also proved within the theorem prover. Any errors detected in the theorem prover can be corrected by changing the corresponding *NDlog* programs. Our initial *DRIVER* prototype uses the PVS theorem prover, due to its substan-

tial support for proof strategies which significantly reduce the time required in the interactive proof process. However, the techniques describe in this paper are agnostic to other theorem provers. We have also verified some of the properties presented in this paper using the Coq [3] proof assistant.

As a second option, *DRIVER* allows the network designer to first utilize a theorem prover to check the protocol design. This requires a network designer first develop formal specifications for the routing protocol of interests. Once the formal representation of the protocol is verified by the prover, corresponding *NDlog* programs are then generated for execution. Similar to the first option, this approach is made possible by the use of *NDlog*, which is particularly amenable to the translation into formal specification recognizable by existing theorem provers (and vice versa), due to its logic-based nature.

Reference [18] provides details on the translation process from *NDlog* programs into formal specification in theorem prover, as well as several verification use cases for standard network routing protocols. In the rest of the paper, we focus on the second approach.

4 Implementing Networks from Verified Specifications

We present the second option for bridging network verification and implementation as described in the previous section. In this approach, a network designer first develops component-based models for their networks. These models are then used to generate formal specifications that can be directly verified in PVS, and the verified PVS specifications can be further compiled into *NDlog* programs for execution.

Our main driving example is based on a component-based model of routing protocols. Given this model, a large family of routing protocols can be implemented simply by customizing subcomponents. The use of components provides the usual benefits of modularity and re-usability. More importantly, in our context, it enables a straightforward translation to formal specifications for verification in PVS or any other general purpose theorem prover.

Our approach of representing and verifying system implementation as components and predicates is adapted from similar techniques in hardware verification [5, 16], where circuits can easily be modeled by composing together electrical components. Interestingly, component-based abstractions have similarly been explored in the networking literature (e.g. [11, 14, 17]), where network protocols are typically composed from existing ones by layered (vertically) or bridging (horizontally) with existing protocols.

4.1 Component-based Model

Before going into the specifics of routing implementations derivation from verified specification, we first provide an introduction to the component-based model adapted from hardware verification, and describe its translation to PVS axioms and equivalent *NDlog* programs. The translation is made possible via the use of *predicative specifications* [5, 16]. In a nutshell, each component is specified as a predicate (relation) over all its external attributes. The external attributes constitute the component's interface, whereas the internal sub-components constitute its implementation in terms of sets of constraints.

To illustrate, we consider an example component `c` shown in Figure 2 (a) with input `I1, I2` and output `O`. This component is implemented in terms of three sub-components `c1, c2, c3`, as shown in Figure 2 (b).

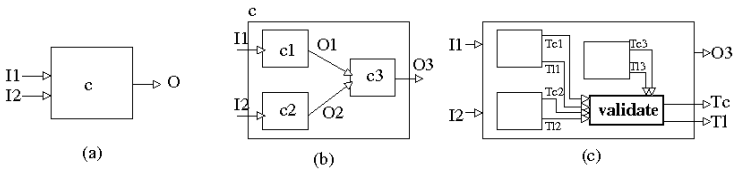

Fig. 2. *Component Representation*

The predicative specification of the corresponding component in PVS is then written as:

```
c(I1,I2,O3): INDUCTIVE bool =
     EXISTS (O1,O2): c1(I1,O1) AND c2(I2,O2) AND c3(O1,O2,O3)

c1(I,O): INDUCTIVE bool = Constraints ct1(I,O)
c2(I,O): INDUCTIVE bool = Constraints ct2(I,O)
c3(I1,I2,O): INDUCTIVE bool = Constraints ct3(I1,I2,O)
```

The top-level component `c` is defined as the conjunction of its sub-components. Within each subcomponents `c1-c3`, there are additional constraints `ct1-ct3` which are typically a conjunction of predicates. We use PVS `Inductive` definition to ensure that all components represent the smallest sets satisfying the constraints in the body.

Translating to Datalog programs: Given the above PVS axioms, there is a straightforward translation to equivalent Datalog programs, by leveraging the proof-theoretic semantics of Datalog. For instance, the following four Datalog rules `t1-t4` implement component `C`:

```
t1 c(I1,I2,O)  :- c1(I1,O1), c2(I2,O2), c3(O1,O2,O3).
t2 c1(I,O)     :- ct1(I,O).
t3 c2(I,O)     :- ct2(I,O).
t4 c3(I1,I2,O) :- ct3(I1,I2,O).
Query c(I1,I2,O).
```

The above translation is feasible as long as each individual set of component constraints (i.e. `ct1`, `ct2` and `ct3`) can be specified as conjunction of set of predefined predicates. Additional location specifiers are supplied by the network designer as part of the model in order to compile the equivalent distributed *NDlog* programs with the correctly annotated location specifiers. Together with the above rules, one can further specify the output of interest in `Query` statement, which in this case is simply the `c` component, i.e. `c(I1,I2,O)`.

Adding soft-state constraints: To support protocol verification in dynamic networks via soft-state semantics as described in Section 2.2, we add two additional attributes `Tc` and `Tl` to each component's interface, as shown in Figure 2 (c). `Tc` and `Tl` denote the creation time and lifetime of each component respectively. The creation times `Tc` are typically a variable in PVS specification, and

lifetimes `Tl` are soft-state lifetimes initialized by the network designer. To illustrate by a concrete example, consider the component `c` introduced earlier. The equivalent PVS specification capturing soft-state semantics is as follows:

```
c(I1,I2,O3,Tc,Tl): INDUCTIVE bool =
EXISTS (O1,O2,Tc1,Tl1): c1(I1,O1,Tc1,Tl1) AND c2(I2,O2,Tc2,Tl2) AND
  c3(O1,O2,O3,Tc3,Tl3) AND validate(Tc,Tl,Tc1,Tl1,Tc2,Tl2,Tc3,Tl3)
```

We further add a `validate` component as a subcomponent to impose the constraints that only components active within the same period can interact with each other. The `validate` component also determines the creation time `Tc` and lifetime `Tl` for the top-level component based on those of sub-components. For example, consider the following PVS specification of a `validate` component:

```
validate(Tc,Tl,Tc1,Tl1,Tc2,Tl2,Tc3,Tl3): INDUCTIVE bool =
Tl=TTL AND Tc1<Tc<=Tc1+Tl1 AND Tc2<Tc<=Tc2+Tl2 AND Tc3<Tc<Tc3+Tl3
  AND Tc=max(Tc1,Tc2,Tc3)
```

The above `validate` subcomponent takes as input the lifetimes of all the other components, and ensures that the creation time `Tc` of the resulting component is set to the max of all creation times among its components, and that `Tl` is set to the specified soft-state lifetime (TTL) of the component `c`. The `validate` also includes additional constraints that ensure that only active components whose lifetimes overlap within the same time window are allowed to interact with each other.

The translation to equivalent Datalog program is straightforward and we omit for brevity.

4.2 Distance Vector Protocol

To demonstrate the component-based model and translation to PVS specifications and *NDlog* programs, we provide a representative example based on the *distance-vector protocol*. This protocol is typically used for *inter-domain* routing, i.e. computing shortest-paths within a local area network or administrative domain (Internet service provider). On the other hand, the path vector protocol presented earlier is generally used for computing routes across over administrative domains. Interestingly, mapping from one protocol to another only require only minor changes to the component specifications.

In the distance vector protocol, each router in the network executing the protocol maintains a routing table, and periodically advertises its current best routes to its neighbors, and updates its routing knowledge when receiving route advertisements from neighbors. Figure 3 shows the component-based model for an instance of the distance-vector protocol in which route advertisements are generated every 5 seconds, and all received routes are stored at each node for 10 seconds to recompute current best hops along shortest paths with minimal hop cost.

The model consists of three main components: `hop`, `hopMsg` and `bestHop` respectively, depicted as **bold** boxes in Figure 3. Besides these three top-level composite components that are built upon sub-components, we introduce a set of *atomic components* depicted by regular (not in bold) boxes, to represent input data (representing a node's prior knowledge of the network) or a pre-defined function (for arithmetic and path concatenation). For example `link` denotes the

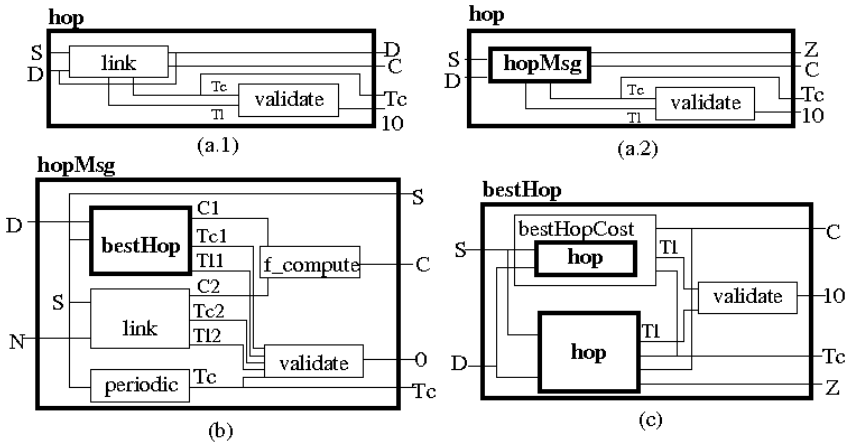

Fig. 3. *Component Representation*

set of neighbors for each node, and `f_compute` is a function computing best route cost.

Figure 3 (a) shows `hop` component with attributes `S,D,Z,C` is used to compute all possible routes from `S` to `D` via next hop `Z` with cost C. The component actually have two instances: the left `hop` component instance (a.1) is derived via a direct one-hop link, whereas in (a.2) the component instance computes hops of increasing hop cost recursively from advertisements received from neighbors (`hopMsg`).

Since hops have associated lifetimes, soft-state attributes described in Section 4.1 are added as additional attributes to each component. The PVS specification for `hop` is as follows:

```
hop(S,D,Z,C,Tc,Tl): INDUCTIVE bool =
  (link(S,D,Tc,10) AND Z=D AND Tl=10 AND C=1) OR
  (EXISTS (C2:Metric): hopMsg(S,D,Z,C2,Tc2) AND Tl=10 AND Tc=Tc2+5)
```

Since route advertisements are triggered every 5 seconds, the additional constraint `Tc=Tc2+5` defined by the `validate` component requires that the creation time of each newly computed `hop` to be advanced accordingly. For ease of exposition, we unfold the `validate` component constraints into the above PVS specification.

Figure 3 (b) shows the `hopMsg` component that denotes a route advertisement. The component's implementation is represented by the conjunction of constraints imposed by all its internal components `bestHop`, `link`, `periodic`, `f_compute` and `validate`. The `periodic` component is a special atomic component `periodic` used to periodically trigger route advertisement at node `S` at time `Tc`. `f_compute` is a function used to compute new route costs from existing route advertisement and `link` costs. The PVS specification for `hopMsg` is as follows:

```
hopMsg(N,D,S,C,Tc,Tl): INDUCTIVE bool =
(EXISTS (Tc2,Tc3,C1,C2:Time): periodic(S,5,Tc) AND
  bestHop(S,D,Z,C1,Tc2,10) AND link(S,N,C2,Tc3,10) AND
  Tc2<Tc<=Tc2+10 AND Tc3<Tc<=Tc3+10 AND C=C1+C2 AND Tl=0)
```

Intuitively it means S will send N a route to reach destination D of cost C via itself if and only if S knows a best route to reach D of cost C1, as indicated by bestHop(S,D,Z,C1,Tc2,10), and that S is N's direct neighbor with a link of cost C1. Note that validate component above sets the creation time of hopMsg as that of the triggering component periodic, and in addition requires all the internal components to be alive during the same window of period.

Finally, Figure 3 (c) shows the bestHop component, which is formalized in PVS as follows:

```
bestHop(S,D,Z,C,Tc,Tl): INDUCTIVE bool =
  bestHopCost(S,D,C,Tc,10) AND hop(S,D,Z,C,Tc,10)
```

For each pair of source S and destination D, the above PVS specifications computes the next hop Z with minimal cost C along the shortest path to the destination. The component utilizes a subcomponent bestHopCost that is used to compute the min aggregation over attribute C of hop to select the best (lowest) cost. We view this aggregation as an atomic service provided by bestHopCost and specify it in PVS as follows:

```
bestHopCost(S,D,MIN_C,Tc,Tl): INDUCTIVE bool =
  (EXISTS (Z:Node): hop(S,D,Z,MIN_C,Tc) AND Tl=10 AND
  (FORALL (C:Metric): (EXISTS (Z:Node): hop(S,D,Z,C,Tc,10))=>MIN_C<=C))
```

Given the above PVS specifications and additional information on location specifiers of predicates, the following *NDlog* program can be automatically generated.

```
dv1 hop(@S,D,D,C,Tc,10):-link(@S,D,C,Tc,10).
dv2 hop(@S,D,Z,C,Tc,10):-hopMsg(@S,D,Z,C,Tc2),Tc=Tc2+5
dv3 bestHopCost(@S,D,min<C>,Tc,10):- hop(@S,D,D,C,Tc,10).
dv4 bestHop(@S,D,Z,C,Tc,10):- bestHopCost(@S,D,C,Tc,10),
                    hop(@S,D,Z,C,Tc1,10), Tc1<Tc<=Tc1+10.
dv5 hopMsg(@N,D,Z,C,Tc,0):- periodic_dv(@S,5,Tc), link(@S,N,C2,Tc2,10),
     bestHop(@S,D,Z,C1,Tc1,10),C=C1+C2,Tc2<Tc<=Tc2+10,Tc1<Tc<=Tc1+10.
```

The above *NDlog* program implements the declarative distance-vector protocol as presented in references [13, 12]. The *min* keyword in rule dv3 is a built-in aggregation construct commonly used in database query languages, and it corresponds to the min computation in the bestHopCost component. The interested reader is referred to these references on detailed performance evaluation of the above declarative protocol using the *P2* declarative networking engine.

4.3 Example Proofs: Convergence and Divergence Analysis

Given the PVS specifications, one can verify a variety of properties of the distance-vector protocol. Assuming distance vector is executed every 5 seconds, and all soft-state predicates have a lifetime of 10 seconds, network convergence can be expressed as:

```
bestHopCost_converge: THEOREM
  EXISTS (j:posnat): FORALL (S,D:Node)(C:Metric)(i:posnat):
    (i>j)=> bestHopCost(S,D,C,5*i,10) = bestHopCost(S,D,C,5*j,10)
```

Given an input network, the distance-vector protocol requires a number of rounds of communication among all nodes, for route advertisements (in the form of hopMsg) to be propagated in the network. In the above theorem, the existential

quantified variable `j` denotes the initial number of periodic intervals (set to be at least the network diameter) required to propagate all route advertisements. The theorem states that for any arbitrary time `i` after `j`, the value of `bestHopCost` converges (i.e. no longer changes).

The distance-vector protocol converges in the static case. However, in a dynamic network with link failure, the protocol can diverge, caused by a well-known problem known as the *count-to-infinity* problem where the protocol diverges in the presence of link failures. In a network of three nodes `a,b,d` with link failure occurred at time 100, divergence is captured by the following theorem:

```
bestHop_count_to_infinity: THEOREM
  FORALL (a,b,d:Node)(t:Time)(c:Metric):(t>100 AND bestHop(a,d,b,c,t,10))
    =>(EXISTS (t':Time)(c':Metric):
        (t'>t AND c'>c AND bestHop(a,d,b,c',t',10)))
```

The theorem above states that the distance vector protocol will diverge after link failure, because the best hop from `a` to `d` will increase indefinitely over time, a symptom of the count-to-infinity problem. Due to space constraints, we omit the proofs of the above theorem. The proof for the convergence case is relatively straightforward. The divergence proofs require us to supply additional axioms that describe link dynamics within a three-node cycle. We have also verified that the count-to-infinity problem exists in a cycle of nodes, and well-known fixes such as the *split-horizon* solution can avoid any two-node cycle, and that this solution is insufficient for preventing count-to-infinity problem in three-node cycle. For a complete list of theorems and proofs, refer to reference [7].

5 Related Work

In addition, we briefly compare the *DRIVER* system with existing work on network protocol verification and development.

Model checking is a collection of algorithmic techniques for checking temporal properties of system instances based on exhaustive state space exploration. Recent significant advances in model checking network protocol implementations include *MaceMC* [10] and *CMC* [8]. Compared to *DRIVER*'s use of theorem proving, these approaches are *sound* as well, but not *complete* in the sense that the large state space persistent in network protocols often prevents complete exploration of the huge system states. They are typically inconclusive and restricted to small network instances and temporal properties.

Classical theorem proving has been used in the past few decades for verification of network protocols [2, 6, 9, 4]. Despite extensive work, this approach is generally restricted to protocol design and standards, and cannot be directly applied to protocol implementation. A high initial investment based on domain expert knowledge is often required to develop the system specifications acceptable by some theorem prover (up to several man-months). Therefore, even after successful proofs in the theorem prover, the actual implementation is not guaranteed to be error-free. *DRIVER* is hence a significant improvement over existing usage of theorem proving [2, 9] which typically require several man-months to develop the system specifications, a step that is reduced to a few hours through the use of declarative networking.

In summary, compared with existing tools, by adopting a theorem-proving based approach that can be integrated with component-based declarative protocol development, *DRIVER* provides a unifying framework that bridges specification, verification, and implementation.

6 Future Work

We are exploring more automatic proof support to make *DRIVER* more approachable to non theorem proving expert. Most general-purpose theorem provers utilize an interactive proof process that requires experience of the proof system of these provers. To ease the user-directed proof construction, we plan to introduce into *DRIVER* network-specific proof strategies by leveraging the PVS built-in proof strategy language [2], hence lowering the barrier for adoption by network designers.

References

1. P2: Declarative Networking System. http://p2.cs.berkeley.edu.
2. PVS Specification and Verification System. http://pvs.csl.sri.com/.
3. The Coq Proof Assistant. http://coq.inria.fr.
4. K. Bhargavan, D. Obradovic, and C. A. Gunter. Formal verification of standards for distance vector routing protocols. *J. ACM*, 49(4):538–576, 2002.
5. A. Camilleri, M. Gordon, and T. Melham. Hardware verification using higher-order logic. Technical Report 91, Computer Laboratory, University of Cambridge, June 1986.
6. R. Cardell-Oliver. On the use of the hol system for protocol verification. In *TPHOLs*, pages 59–62, 1991.
7. DNV use cases for protocol verification. http://www.seas.upenn.edu/∼anduo/dnv.html.
8. D. Engler and M. Musuvathi. Model-checking large network protocol implementations. In *NSDI*, 2004.
9. A. P. Felty, D. J. Howe, and F. A. Stomp. Protocol verification in nuprl. In *CAV*. Springer-Verlag, 1998.
10. C. Killian, J. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI*, 2007.
11. E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.
12. B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *ACM SOSP*, 2005.
13. B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *ACM SIGCOMM*, 2005.
14. Y. Mao, B. T. Loo, Z. Ives, and J. M. Smith. MOSAIC: Unified Platform for Dynamic Overlay Selection and Composition. In *CoNEXT*, 2008.
15. S. Raman and S. McCanne. A model, analysis, and protocol framework for soft state-based communication. In *SIGCOMM*, pages 15–25, 1999.
16. M. Srivas, H. Rueß, and D. Cyrluk. Hardware verification using PVS. 1997.
17. The Ensemble Distributed Communication System. http://dsl.cs.technion.ac.il/projects/Ensemble/.
18. A. Wang, P. Basu, B. T. Loo, and O. Sokolsky. Declarative Network Verification. In *11th International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2009.

Formalizing Metarouting in PVS

Anduo Wang[1] Boon Thau Loo[1]

University of Pennsylvania
{anduo,boonloo}@seas.upenn.edu

Abstract. In this paper, we extend PVS specification logic with abstract *metarouting* theory to aid the development of complex routing protocol models based on *metarouting*, which is an algebraic framework for specifying routing protocols in a restricted fashion such that the protocol is guaranteed to converge. Our formalization of metarouting theory utilizes the *theory-interpretation* extensions of PVS. Our use of a general purpose theorem prover provides a structured framework for a network designer to incrementally develop and refine their algebraic routing protocol model by starting from various base routing algebras, and composing them into complex algebra models with composition operators. In addition, one can leverage PVS's type checking capability and built-in proof engine to ensure routing model consistency.

1 Introduction

The Internet today runs on a complex routing protocol called the *Border Gateway Protocol* or *BGP* for short. BGP enables Internet-service providers (ISP) world-wide to exchange reachability information to destinations over the Internet, and simultaneously, each ISP acts as an autonomous system that imposes its own import and export policies on route advertisements exchanged among neighboring ISPs.

Over the past few years, there has been a growing consensus on the complexity and fragility of BGP routing. Even when the basic routing protocol converge, conflicting policy decisions among different ISPs have lead to route oscillation and slow convergence. Several empirical studies such as [7] have shown that there are prolonged periods in which the Internet cannot reliably route data packets to specific destinations due to routing errors induced by BGP. In response, the networking community has proposed several Internet architectures and policy mechanisms (e.g. [1]) aimed at addressing these challenges.

Given the proliferation of proposed techniques, there is a growing interest in formal software tools and programming frameworks that can facilitate the design, implementation, and verification of routing protocols. These proposals can be broadly classified as: (1) algebraic and logic frameworks (e.g. [3]) that enable protocol correctness checking in the design phase; (2) runtime debugging platforms that provide mechanisms for runtime verification and distributed replay, and (3) programming frameworks that enable network protocols to be specified, implemented, and in the case of the Mace toolkit, verified via model checking [6].

In this paper, we extend PVS specification logic with abstract *metarouting* theory [3] to aid the development of complex routing protocol models based on *metarouting*, which is an algebraic framework for specifying routing protocols in a restricted fashion such that the protocol is guaranteed to converge. Using

the *theory-interpretation* [8] extensions of the PVS theorem prover, we formalize in PVS a variety of metarouting algebra instances and demonstrate that an interactive theorem prover is suitable for modeling the complicated BGP system using the metarouting theory developed in PVS.

The main benefits of formalizing metarouting within a mechanized theorem prover are as follows. First, the network designer can now focus on high-level protocol design and the conceptual decomposition of the BGP system, and shift the low level details of ensuring consistency of the derived protocol model with respect to metarouting theory to the PVS type checker. Second, the PVS proof engine handles most of the proof effort (via the top-level strategy *grind* and other built-in type checking capabilities), and therefore frees the network operator from the trivial and tedious proof necessary to ensure the convergence of their BGP algebra model. In the long run, we believe that our framework will also result in the support of relaxed algebra models, which allow a wider range of well-behaved convergent component protocols to be supported compared to the restrictions imposed by metarouting.

2 Background

2.1 Internet Routing

The Internet can be viewed as a network of *Autonomous Systems* (AS) each administrated by an *Internet Server Provider* (ISP). The *routing protocol* is executed on all ASes in order to compute reachability information. Given a destination address, each packet sent by a source is forwarded by each intermediate node to the next neighboring node along the best path computed by the routing protocol.

In particular, within an AS, the ISP runs its own class of routing protocols called the *Internal Gateway Protocol* (IGP), whereas between ASes , the class of protocols used are called the *External Gateway Protocol* (EGP). EGP enables routing across AS administration borders by including mechanism for *policy-based routing*. The role of policy routing is to allow ISPs to influence route decisions for economical or political concerns, and the basic mechanism used is to decide which routes to accept from neighbors (*import policies*), and which routes to advertise to other neighbors (*export policies*).

2.2 Metarouting

The Internet uses the *Border Gateway Protocol* (BGP) as its de facto routing protocol. This protocol is a combination of the IGP/EGP protocol described above. Metarouting [3] is first proposed to extend the use of routing algebra to BGP design and specification. Metarouting enables the construction of a complicated BGP system model from a set of pre-defined base routing algebras and composition operators. Prior to metarouting, Griffin *et al.*first proposed combinatorial models for BGP [2, 4] to aid the static analysis of convergence of routing protocols. Later a *Routing Algebra Framework* was proposed by Sobrinho [9, 10] to provide the rigorous semantics for the design and specification of routing protocols. Sobrinho uses various algebra instances to represent possible routing protocols and policy guidelines. Sobrinho further identifies and proves monotonicity as a sufficient condition for protocol convergence. Meta-routing builds upon these two earlier pieces of work. In the rest of the section, we provide a short overview of metarouting.

First, metarouting adopts the use of routing algebra as the mathematical model for routing. An abstract routing algebra is a tuple A: $A = \langle \Sigma, \preceq, \mathcal{L}, \oplus, \mathcal{O}, \phi \rangle$. Here Σ is the set of *signatures* used to describe paths in the network totally ordered by preference relation $\preceq$. Intuitively, the preference relation is used by a routing protocol to optimize path cost; $\mathcal{L}$ is a set of *labels* describing links between immediate neighbors. Note that labels may denote complicated policies associated with the corresponding link; $\oplus$ is a mapping from $\mathcal{L} \times \Sigma$ to Σ, which is the *label application operation* that generates new paths by combining existing paths and adjacent links; And $\mathcal{O}$ is a subset of Σ called *origination* that represents the initial routes stored at network nodes; Finally ϕ is a special element in Σ denoting the prohibited path. The semantics of routing algebra is given by the following axioms:

$$\begin{aligned}
\textbf{Maximality} \quad & \forall_{\alpha \in \Sigma - \{\phi\}} \quad \alpha \preceq \phi \\
\textbf{Absorption} \quad & \forall_{l \in \mathcal{L}} \quad l \oplus \phi = \phi \\
\textbf{Monotonicity} \quad & \forall_{l \in \mathcal{L}} \forall_{\alpha \in \Sigma} \quad \alpha \preceq l \oplus \alpha \\
\textbf{Isotonicity} \quad & \forall_{l \in \mathcal{L}} \forall_{\alpha, \beta \in \Sigma} \quad \alpha \preceq \beta \implies l \oplus \alpha \preceq l \oplus \beta
\end{aligned}$$

Maximality and **Absorption** are straightforward properties of the prohibited path ϕ, stating that any other paths are always preferred over ϕ, and that extending the un-usable path ϕ with any usable link would still result in prohibited path. On the other hand, **Monotonicity** and **Isotonicity** are two non-trivial properties that ensure network *convergence*[1] of a routing protocol modeled by the routing algebra.

Furthermore, based on the abstract routing algebra, metarouting identifies a set of atomic (base) algebras such as *ADD(n,m)* and *LP(n)*, and composition operators such as *Lexical Product* $\otimes$ and *Scaled Product* $\odot$ as the building blocks for more complicated routing algebras. This paper presents the incremental development of metarouting abstract algebras and the use of such abstract theory to build concrete BGP systems.

Unlike previous combinatorial models [2, 4], metarouting identifies and proves that the properties of *monotonicity* and *isotonicity* are sufficient conditions for network convergence. Convergence verification of BGP systems are then reduced to proofs of monotonicity and isotonicity of the related routing algebra, whereas in the analysis of BGP systems using previous combinatorial models, the proof requires genuine insights into the models themselves.

Despite its advantages, metarouting is fairly restricted in two ways. First, it cannot represent all protocols that converge. Second, it places the burden on network designers to write algebras and composition operators correctly. Our work aims to address these two limitations by using PVS to provide a framework for expressing routing algebras and their operators correctly, and then flexibly reason about the convergence properties of these protocols even when the sufficient conditions are violated. One should view our paper as providing the initial building blocks and methodology for interesting explorations elaborated in Section 5.

[1] A network routing protocol converges when all routing tables can be computed to a distributed fixpoint given a stable network, and when any links are updated, these routing tables can be incrementally recomputed similarly to a fixpoint.

3 Basic Approach

This section describes the basic technique of embedding metarouting in PVS. In particular, this paper presents the development of metarouting in PVS using its extensions on *theory interpretation* [8].

The basic approach is to encode metarouting algebraic objects in PVS's type system. It involves formalization of abstract routing algebra theory A, the set of atomic algebra instances of A, and composition operator $\otimes$.

First of all, the abstract routing algebra structure $A = \langle \Sigma, \preceq, \mathcal{L}, \oplus, \mathcal{O}, \phi \rangle$ is formalized as an uninterpreted abstract (source) theory in PVS, as described in [11, 5, 8]. The basic idea is to use *types* to denote the sets of objects $\Sigma, \mathcal{L}, \mathcal{O}$. Accordingly, the special element ϕ denoting prohibited path is expressed as an uninterpreted constant of type Σ. And the preference relation $\preceq$ and the label application operation are denoted by functions. The PVS theory for abstract route algebra A is given as follows:

```
routeAlgebra: THEORY
  BEGIN
  sig: TYPE+
  prefRel: [sig, sig -> bool]
  label: TYPE+
  labelApply: [label, sig -> sig]
  prohibitPath: sig
  initialS: [sig -> bool]
  org: TYPE = s: sig | initialS (s)
  END routeAlgebra
```

Here uninterpreted types `sig` and `label` denote sets Σ and $\mathcal{L}$, and `org` denoting initial route set $\mathcal{O}$ is made subtype of Σ via auxiliary predicate `initialS` which decides if a path falls into the initial routes. Finally ϕ is made constant `prohibitPath` of type `sig`.

Semantics of abstract routing algebra A are given by the following axiomatic specification:

```
monotonicity: AXIOM FORALL (l: label, s: sig): mono(l, s)
isotonicity: AXIOM
  FORALL (l: label, s1, sig, s2: sig):
    prefRel(s1,s2) => prefRel(labelApply (l,s1), labelApply(l,s2))
maximality: AXIOM FORALL (s: sig): prefRel (s, prohibitPath)
absorption: AXIOM
  FORALL (l: label): labelApply (l, prohibitPath) = prohibitPath
```

Where `eqRel` (`eqRel` will be used later) and `mono` are defined by the following auxiliary predicates:

```
  eqRel (s1, s2: sig): bool = prefRel (s1, s2) and prefRel (s2, s1)
  mono(l: label, s: sig): bool = prefRel (s, labelApply (l, s))
```

Note that this abstract routing theory A then stands for all possible routing algebra instances. We also observed that PVS parametric theories offer an alternative to define abstract algebra, sketched as follows:

```
routeAlgebra[sig: TYPE+, prefRel: [sig, sig -> bool],
  label:TYPE+, labelApply: [label, sig -> sig]]
 BEGIN
  prohibitPath: sig
  initialL: [label -> bool]
  org: TYPE+ = l: label | initialL (l) ...
```

In the rest of this paper, to exploit PVS's theory interpretation mechanism, we will use of the first uninterpreted theory representation. To encode the basic building blocks of metarouting: atomic routing algebras and composition operators, we utilize two features provided by the PVS theory interpretation [8] extensions: *mapping* and *declaration*. With the *mapping* mechanism, a general (source) theory is instantiated to an interpretation (target) theory. On the other hand, the *theory declaration* mechanism takes PVS theories as parameters, and therefore, unlike mapping, can build a theory from multiple source structures.

Figure 1 shows our basic two-step approach to formalize metarouting building blocks. First, we utilize abstract routing algebra theory `A` developed above as a source theory, applying PVS's *mapping* mechanism to this source theory to yield the set of interpretation theories `Ii` for atomic routing algebras I_i. Second, by applying PVS's theory *declaration* mechanism, we encode composition operator O_i as PVS theories taking routing algebra as parameters, which can be further instantiated to yield the resulting compositional routing algebra `Oi`.

The main benefit of this approach is that the semantics (axioms) of source routing theory `A` are enforced automatically in all target theories `Ii` and `Oi`. This ensures that all atomic routing algebra instances are valid routing algebras and that all composition operators are closed under abstract routing algebra (i.e. any compositional routing algebras that can be derived using operators `Oi` are guaranteed to be routing algebras as defined by the abstract routing algebra theory). The detailed formalization of metarouting building blocks using PVS theory interpretation is presented in the next section.

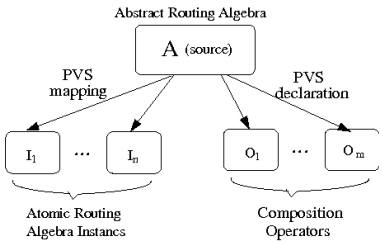

Fig. 1. Overview of PVS Theories

4 Compositional Routing Algebra

This section presents the formalization of metarouting building blocks by stepping through atomic routing algebras *addA(n,m)* and *cpA(n)*; as well as composition operator lexical product $\otimes$.

4.1 Atomic Routing Algebra Instance

Shortest Path Routing The first simple routing algebra *addA(n,m)* describes shortest path routing. The labels/signatures can be thought of as the distance costs associated with the corresponding links/paths. Note that in practice, costs of valid links/paths have an upper bound, and links/paths with higher cost are considered prohibited. We use PVS theory `addA` to capture algebra *addA(n,m)* as follows:

```
addA: THEORY
 BEGIN
  n, m: posnat
  N_M: AXIOM n < m
  LABEL: TYPE = upto(n)
  SIG: TYPE = upto(m + 1) ...
```

Here `n, m` are the uninterpreted constants denoting link/path cost bounds. The preference relation $\preceq$ over signatures `SIG` are then simply interpreted as the normal $\leq$ relation over natural numbers, indicating that a low-cost path is preferred over high-cost path. The label application operation $\oplus$ can be interpreted as a function that computes the cost of the new path obtained from a sub-path and the adjacent link, where the cost of the new path is simply the normal addition of that of the sub-path and link. In PVS, we write as follows:

```
PREF(s1, s2: SIG): bool = (s1 <= s2)
APPLY(l: LABEL, s: SIG): SIG = IF (l+s < m+1) THEN (l+s) ELSE (m+1) ENDIF
```

Note that in the definition of label application function `APPLY`, `m+1` is used as the value of prohibited path. This is directly defined using PVS mapping of abstract algebra `routeAlgebra` in the following `IMPORTING` clause:

```
IMPORTING routeAlgebra{{sig := SIG, label := LABEL,
                        prohibitPath := m + 1,
                        labelApply(l:LABEL, s:SIG) := APPLY(l,s),
                        prefRel(s1, s2: SIG) := PREF (s1, s2)}}
```

Recall that a routing algebra consists of a set of signatures `sig`, labels `label`, preference relations `prefRel` over signatures, and label application functions `labelApply`. Here, theory `addA` imports the uninterpreted abstract algebra theory `routeAlgebra`, and makes the following instantiations:

$$\begin{aligned} \texttt{sig} &\leftarrow \texttt{upto(n)} \\ \texttt{label} &\leftarrow \texttt{upto(m)} \\ \texttt{prohibitPath} &\leftarrow \texttt{m} + 1 \\ \texttt{labelApply} &\leftarrow \texttt{APPLY} \\ \texttt{prefRel} &\leftarrow \texttt{PREF} \end{aligned}$$

The corresponding instances of `routeAlgebra` axioms defining the semantics of routing algebra are proof obligations called type correctness conditions (*TCCs*).

For example, monotonicity axiom is instantiated and denoted by the following automatically generated TCC:

```
IMP_A_monotonicity_TCC1: OBLIGATION FORALL (l: LABEL, s: SIG): mono(l, s)
```

All of the TCCs are automatically discharged by either the default TCC proof strategy or high-level strategy `grind`.

So far, we have established the encoding of shortest path algebra in PVS by providing mappings for uninterpreted types in the source theory `routeAlgebra` into the target theory (interpreting) `addA`. We observe that even in this simple example, PVS significantly reduces manual effort ensuring consistency, generating proof obligations and enabling the user to focus on high-level mapping for shortest path routing.

Customer-Provider and Peer-Peer Relationship We provide another example of base/atomic algebra *cpA(n)* that captures the policy guideline regarding the economic relationship between ASes. Customer-Provider and Peer-Peer relationships between ASes are prevalent in today's Internet. A common policy guideline to help BGP convergence is to always prefer customer-routes to peers or providers routes.

More specifically, in the algebra $cpA = \langle \Sigma, \preceq, \mathcal{L}, \oplus, \mathcal{O}, \phi \rangle$, the signature set can take three values $C/R/P$, representing customer/peer/provider routes respectively (i.e. routes advertised by a node's customer, peer, or provider). Accordingly, labels can take values $c/r/p$, representing customer/peer/provider link (i.e. links to customer/peer/provider).

The preference relation over signatures is given by: $C \preceq R, \quad R \preceq P, \quad C \preceq P$. Intuitively this relation means, a customer route is always preferred over a peer and provider route, and a peer route is preferred over a provider route. The intuition is that each ISP enforces the policy to reduce the use of provider routes, while maximizing availability and use of its customer routes.

The complete definition of the label application operation $\oplus$ is given by the following table:

$\oplus$	C	R	P
c	C	C	C
r	R	R	R
p	P	P	P

For example the first line $c \oplus (C/R/P) = C$ can be read as a customer/peer/provider path extended by a customer link results in a customer path, hence has the highest priority of all available paths.

For simplicity, rename labels and signatures as follows: $c \leftarrow 1, r \leftarrow 2, p \leftarrow 3$ and $C \leftarrow 1, R \leftarrow 2, P \leftarrow 3$. This renaming enables the preference relation to be expressed as normal $\leq$ over natural number. Similar to the algebra for shortest path, `cpA` can be encoded using PVS mapping as follows:

```
cpA: THEORY
  BEGIN
  SIG: TYPE = x: posnat | x<=3
  LABEL: TYPE = x: posnat | x<=3
  APPLY (l: LABEL, s: SIG): SIG = l
  IMPORTING routeAlgebra{{sig := SIG, label := LABEL,
                          labelApply (l: LABEL, s: SIG):= APPLY (l,s),
                          prohibitPath := c+1}}
  END cpA
```

As in the case of shortest paths, all the TCCs enforcing routing algebra axioms (for example, monotonicity) are automatically discharged. This is consistent with the intuition that customer-provider policy does help BGP convergence.

4.2 Lexical Product and Route Selection

This section presents development of lexical product $\otimes$, a composition operator that enables construction of routing algebra from atomic algebra described in section 4.1. It is particularly useful in modeling route selection in BGP system where multiple attributes are involved.

Consider product algebras $A \otimes B$ constructed from two route algebra `A, B`, where the parameter theories `A` and `B` model two attributes `a` and `b` respectively. First define the signature and label of $A \otimes B$ as product of that from `A` and `B` in PVS as:

```
lexProduct[A, B: THEORY routeAlgebra]: THEORY
  BEGIN
  SIG: TYPE = [A.sig, B.sig]
  LABEL: TYPE = [A.label, B.label] ...
```

Here the first component of signature/label comes from `A` and the second component comes from `B`. And a natural interpretation of label application function over path and label is given by the following product in PVS:

```
APPLY(l:LABEL,s:SIG):SIG =
(A.labelApply(l‘1,s‘1),B.labelApply(l‘2,s‘2))
```

Here the two components invoke the corresponding label application functions defined in theory `A` and `B` respectively.

Next consider the preference relation over $A \otimes B$ that in PVS as follows:

```
PREF(s1,s2:SIG):bool = A.prefRel(s1‘1, s2‘1) OR
                       (A.eqRel(s1‘1,s2‘1) AND B.prefRel(s1‘2,s2‘2))
```

The above definition is particularly interesting because it models the route selection process in BGP system. This preference relation reads as: a path with two attributes `a` and `b` represented by signature `s1` is considered better than a path denoted by `s2` given one of the two following conditions: (1) first component `s1‘1` of `s1` is better than the first component `s2‘1` of `s2`, as defined in algebra `A`; or (2) if the first component of `s1` and `s2` are equally good, but `s1` is better than `s2` with respect to the second component, as described in algebra `B`. This lexicographic comparison captures the route selection process, which is a major part for any BGP system with multiple attributes. Intuitively, in selecting a route towards a given destination, the router compares all its possible paths towards that destination by going through a comparison list, checking one attribute at a time, selecting the best path based on attributes ordering. The router goes down the list and compares the next attribute only if the attributes seen in previous steps are equally good.

As before, we can now instantiate route algebra theories and corresponding sets of axioms as follows:

```
IMPORTING routeAlgebra{{sig := SIG, label := LABEL,
                        labelApply(l:LABEL,s:SIG) := APPLY(l,s),
                        prefRel(s1, s2: SIG) := PREF(s1, s2)}}
```

Again, PVS automatically generate and prove all the type checking conditions.

4.3 A Concrete First Example

This section presents an concrete example routing protocol algebra built from metarouting atomic algebras and composition operators developed in previous sections. We demonstrate the ease of applying abstract metarouting theory to concrete example algebra in PVS. In particular, we highlight the intuitive networking interpretation in practice.

Consider a simple BGP system where the route paths are measured in terms of customer-provider relationship and distance cost. For all possible routes reaching a given destination, a route path going through customers and peers is preferred to path going through providers; and a route go through peers is preferred to those through providers. Once this customer-provider policy is enforced, the ISP is concerned with distance cost with respect to each path. For the same types of paths, the ISP will choose the shortest path with lowest cost.

In the top level, this BGP system can be decomposed into two sub-components: customer-provider component and the shortest path component developed in section 4.1. Because the customer-provider relationship has higher-priority over the distance cost attribute, it can be naturally implemented by construction using lexical product, as shown in the following PVS code:

```
firstExample: THEORY
 BEGIN
 IMPORTING AlgebraInstance, lexProduct
 firstAlgebra: THEORY = lexProduct[A2,B2]
```

Here `firstAlgebra` is defined to be the concrete algebra modeling this BGP system. It is constructed from customer-provider component algebra `A2` and shortest path algebra `B2` by applying lexical product, where `A2` and `B2` are defined in the imported theory `AlgebraInstance`. First, we show the definition of `A2` that enforces ISP customer-provider policy simply as an instance of `cpA`, where the uninterpreted constant `c` is mapped to 3.

```
AlgebraInstance: THEORY
 BEGIN
  IMPORTING cpA{{ c := 3 }}
  A2:THEORY = routeAlgebra{{sig = cpA.SIG, label = cpA.LABEL,
                            labelApply(l:cpA.LABEL,s:cpA.SIG) = mod(l+s,c),
                            prohibitPath = c + 1,
                            prefRel(s1, s2: cpA.SIG) = (s1 <= s2) }} ...
```

Likewise, concrete algebra `B2` for shortest path can be defined in terms of `addA` as follows:

```
IMPORTING addA{{n:= 16, m:=16}}
B2:THEORY = routeAlgebra{{sig = addA.SIG, label = addA.LABEL,
                          labelApply(l:int,s:int) = l+s,
                          prohibitPath=16, prefRel(s1,s2:int) = (s1<=s2)}}
```

Where uninterpreted bounds on signature/labels `m/n` in `addA` are mapped to 16, which is the actual value used in distance vector protocol practice. Finally, by type checking, PVS automatically figures out all type correctness conditions to ensure consistency. All of the TCCs are discharged with default/high-level proof procedure in one step. This ensures the BGP system we derived from atomic algebras `addA, cpA` by using composition operator $\otimes$ are indeed a valid routing algebra that is guaranteed to converge.

In summary, we observe that by incorporating metarouting abstract theory, a non-specialized standard proof assistant like PVS, can be used to specify a specific routing protocol instance with great ease. And the routing algebra semantics is enforced by proof obligations (TCCs) automatically generated in PVS, all of which can be discharged by either PVS default TCC proof strategy or high-level strategy `grind` in one step!

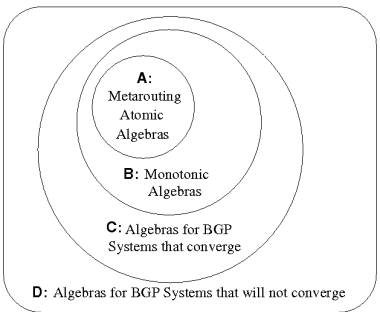

Fig. 2. Classification of BGP algebras

5 Future Work

A straightforward application of the specification technique we explored in this paper is to construct incrementally in PVS the routing algebraic model for complicated BGP systems by using the base algebra blocks and composition operators we developed in this paper, and the resulting algebraic model checked

in PVS is then used as part of design document to derive the real BGP implementation. To achieve full set support for the modeling of BGP system via metarouting, we plan to encode in PVS more base routing algebras, such as `TAG` which is critical in the modeling of complicated routing policies, and more composition operators, such as scoped product, which models a BGP system running in and between administrative regions (i.e. the behavior of BGP protocol across AS boarder). Furthermore, we conceive a more ambitious (adventurous) use of PVS to aid the verification of BGP system convergence using a relaxed algebra model. As depicted in Figure 2, we label the set of atomic metarouting algebras with type `A` and denote them with the inner ring. We then observed that all metarouting algebras that can be composed from type `A` algebras by composition satisfy monotonicity by definition and therefore fall into type `B` algebras represented by the middle ring. Sobrinho's original paper [9] showed that monotonicity is a *sufficient* (not necessary) condition for BGP system convergence. There are known BGP systems that converge but violate monotonicity, and this reveals existence of type `C` algebras modeling the set of converging BGP systems that are not monotonic. By relaxing the monotonicity property, we would like to explore the modeling and reasoning of type `C` systems that fall outside Monotonic type `B` Algebra (the middle ring) but are equally good with respect to convergence. Taking this basic approach one step further, instead of starting from the algebra model, we would like to develop in PVS an algebraic representation of a given BGP system that falls outside the scope of current metarouting algebra, and with the aid of PVS proof engine, decide if that corresponding BGP system falls into type `C` and converges or type `D` that does not converge.

References

1. C. T. Ee, B.-G. Chun, V. Ramachandran, K. Lakshminarayanan, and S. Shenker. Resolving Inter-Domain Policy Disputes. In *SIGCOMM*, 2007.
2. T. G. Griffin, F. B. Shepherd, and G. Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Trans. Netw.*, 2002.
3. T. G. Griffin and J. L. Sobrinho. Metarouting. In *ACM SIGCOMM*, 2005.
4. T. G. Griffin and G. Wilfong. An Analysis of BGP Convergence Properties. *SIGCOMM Comput. Commun. Rev.*, pages 277–288, 1999.
5. E. L. Gunter. Doing algebra in simple type theory. 1989.
6. C. Killian, J. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI*, 2007.
7. C. Labovitz, G. Malan, and F. Jahanian. Internet Routing Instability. *ACM/IEEE Trans. on Networking*, 1998.
8. S. Owre and N. Shankar. Theory interpretations in pvs. Technical report, 2001.
9. J. Sobrinho. Network routing with path vector protocols: theory and applications. In *SIGCOMM*, 2003.
10. J. Sobrinho. An algebraic theory of dynamic network routing. Technical report, October 2005. (William R. Bennett Prize 2006).
11. P. J. Windley. Abstract theories in hol. In *HOL'92: Proceedings of the IFIP TC10/WG10.2 Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 197–210. North-Holland/Elsevier, 1993.

Verifying Compiling Optimisations Using Isabelle/HOL

Richard Warburton and Sara Kalvala

Department of Computer Science
University of Warwick

Abstract. This paper is based on an approach to program optimisation based on transformations, where temporal logic is used to specify side conditions. We present an embedding of this domain specific language in Isabelle/HOL using Constant Propagation as a running example and describe how proofs of soundness can be constructed for given specifications.

1 Introduction

Significant effort in modern compiler development is spent implementing optimisations, which are often performed after an initial synthesis of low-level code [1,2,5]. Global optimisations that exploit complex chains of information and subtle patterns of execution can offer significant rewards in terms of improved program performance. Ensuring that these optimisations don't introduce bugs into the program being optimised by the compiler is a complicated task. Notably one's intuition about the correctness of the program being optimised may not be trustworthy. Currently commercial compilers address this issue with a large testsuite of programs with a simple and known input/output relation that their compiler is tested on. We address the problem differently: to focus on the optimisations themselves and perform machine checked theorem proving in order to ensure that they don't alter the semantics of the program.

The specifications are presented in the TRANS specification language, that uses succinct rewrite rules to transform the program graph, and uses Computational Tree Logic as the basis for its side condition specifications.

2 Background

2.1 Validation and Verification

Formal methods research in compiler correctness is categorisable into two main categories. One approach, known as Translation Validation [7], checks that the input program and output program are semantically equivalent, for a given run of the compiler. In other words, for some definition of equivalence, and where P and P' are the source and transformed programs respectively that $\lceil P \rceil = \lceil P' \rceil$. This is affected by establishing certain checkable criteria that the compiler

must meet after optimisation and instrumenting the compiler to perform the checks. Frequently Automated Theorem Provers are used in order to perform the checking for a given run.

We seek to verify the soundness of a optimisation once and for all. So for some notion of equivalence, and a given optimisation O, we want to show that $\forall PP'.O(P) = P' \rightarrow \lceil P \rceil = \lceil P' \rceil$.

2.2 TRANS

In the TRANS language [4], the optimisations are represented through two components: a rewrite rule and a side condition which indicates the situations in which the rewrite can be applied safely.

The rewrite rules use standard programming syntax (assignment statements, go-to and if statements, etc). They match a sequence of statements within a program and replace them by another. The syntax is expanded with a few constructs to support meta-variables, representing either syntactic fragments of the program or nodes of the CFG.

The side condition language is an extension of first order CTL, where formulae are built up from basic predicates that describe properties of states. There are two types of these basic predicates used to obtain information about a node in the control flow graph; these are the *node* and *stmt* predicates. The formula $node(x)$ will hold at a node n in a valuation that maps n to x. The formula $stmt(s)$ will hold at a node n where the valuation makes the pattern s match the statement at node n. As well as judgements about states the language can make "global" judgements. For example, the formula $\phi \ @ \ n \ \wedge conlit(c)$ states that ϕ holds at n and c is a constant literal throughout the program.

A logical judgement of the form: $\phi \ @ \ n$ states that the formula ϕ is *satisfied* at node n of the control flow graph. We base our language for expressing conditions on CTL [3], a path-based logic which can express many optimisations while still being efficient to model-check. However, we modify the logic slightly to make it easier to express properties of programs: we include past temporal operators ($\overleftarrow{E}$ and $\overleftarrow{A}$) and extend the next state operators (EX and AX) so that one can specify what kind of edge they operate over. For example, the operators EX_{seq} and AX_{branch} stand for "there exists a next state via a *seq* edge" and "for all next states reached via a *branch* edge" respectively.

It is also possible to make use of user defined predicates via a simple macro system. These can be used in the same way as core language predicates such as *use*. They are defined by an equality between a named binding and the temporal logic condition that the predicate should be expanded into.

[4] demonstrates the expressiveness of TRANS by presenting a catalog of optimisations including lazy code motion, constant propagation, strength reduction, branch elimination, skip elimination, loop fusion, dead code elimination and lazy strength reduction.

[8] provides an approach to generating compiler optimisations from TRANS specification. These optimisations can be applied to `Java` bytecode, using the Soot framework.

2.3 Jinja

We base our model of program semantics on the Jinja language [6], which is a Java like system, entirely mechanized in Isabelle/HOL. It has been published within the Archive of Formal Proofs *REF*. The theory itself presents big-step and small-step operational semantics for the source language, a proof of equivalence between these semantics, bytecode semantics and a multi-stage compiler from the source language to the bytecode language.

Our use of the Jinja framework focusses on the small-step operational semantics for the source program. The TRANS language's expression pattern matching works at a higher level than bytecode. For example, there being little structural restriction within bytecode, assuming some bytecode sequence passes bytecode verification, one can introduce goto instructions in the middle of integer arithmetic. Additionally arithmetic is compiled into several stack manipulating operations. TRANS makes assumptions about the structure of the language that make bytecode unsuitable. Proofs using TRANS are frequently constructed by proving that some property about program state is implied by a side condition and that that property implies the soundness of a rewrite. This makes small-step semantics more appropriate than big step semantics, since it makes the internal state of the program at some point in its execution more explicit.

Jinja does not completely represent Java; for example, whilst Java has multiple primitive datatypes, such as float and short, Jinja only has an int type. Furthermore the only numerical operations defined within Jinja are addition and comparison. This somewhat limits certain optimisations, for example the strength reduction of multiplication operations into addition operations within loop inductive variables. This can be formally specified within hand written TRANS specifications, but the Isabelle formalisation doesn't support it, since there is no multiplication.

3 An Example - Constant Propagation

For the remainder of this paper we shall use Constant Propagation as a simple example optimisation. Constant Propagation doesn't cover all concepts within the TRANS language, or our framework, but it illustrates key points about the mechanisation and proof technique.

Constant propagation is a transformation where the use of a variable is replaced with the use of a constant known at compile time. An example application to a trivial program is given in Fig. 1.

```
a = 3;          a = 3;
b = a;          b = 3;
```

Fig. 1. Program before and after Constant Propagation

Hence, where a variable `x` is assigned to a variable `v`, replace it with an assignment of a constant `c` to the variable `x`. We consequently specify the rewrite rule as:

$$n : (x := v) \implies x := c$$

The side condition needs to check that `x` will always be assigned to `v` at that program point and that `v` is always assigned to `c` on a path through the control flow graph where this optimisation is applied. It is formulated by looking backwards on all paths through the program until it finds the point at which `v` is assigned to `c`, and checking that `v` is not redefined between there and program point n. Additionally we need another predicate to ensure that `c` is genuinely a constant. A complete specification is given in Fig. 2.

$$\begin{array}{l} n : (x := e[v]) \Longrightarrow x := e[c] \\ \text{if} \\ \quad \overleftarrow{A}(\neg def(v)\ U\ stmt(v := c))\ @\ n\ \wedge\ conlit(c) \end{array}$$

Fig. 2. Specification of constant propagation

4 Mechanisation and Proofs

We attempt to make our embedding as lightweight and shallow as possible. We translate transformations into functions that change expressions within the `Jinja` semantics. It is worth noting that, because of the way that `Jinja` is formalised, there is no distinct notion of a statement, so the body of any method is an expression. Since our framework is only designed to handle intra-procedural optimisation, mutating expressions is all that is required.

4.1 Refinement

The TRANS specifications undergo some hand refinement, from the form that they are written in, to the form that they are used in Isabelle. The rewrite rules are converted into a pattern matching component that forms part of the side condition, and a replacement rule that becomes the new action. The replacement function simply replaces one expression with another in the `Jinja` expression. It is consequently simpler to reason about, and is discussed in section 4.5. Additionally macros are expanded.

The refinement operations are similar to, albeit simpler than, those performed in Rosser system for generating Java bytecode optimisers from the TRANS language, [8] and within that system this process has been fully automated. This is currently future work for the Isabelle formalisation.

4.2 Syntax

A series of datatypes are used in order to embed the syntax of the TRANS language into Isabelle/HOL. For example, the following definition describes some of the expression pattern matching used in TRANS.

datatype
pattern = *Skip* (;)
| *Assign literal patternExpr* (**infixr** := *65*)
| *If patternExpr patternExpr* (*if*)
| *Return patternExpr* (*return*)

We haven't completely formalised the syntax of the TRANS language, and more details are available in **REF**.

4.3 Predicates

Predicates within TRANS can be considered in two ways, firstly there's the property about the program structure or state that is implied by the predicate holding at a certain point in the program, for a certain path through its execution. We call this the semantic property. There is also the function that a compiler author would write in order to implement the corresponding predicate within their system. This implementation function should always imply the semantic property of the predicate, but due to some limitation, or design choice, it may also restrict the program additionally. For example we refer to `use` and `def` predicates, but a compiler implementor in a `Java`-like language would have to conservatively approximate such properties using a `may-use` and `may-def` predicate.

Currently within our system we are translating predicates into their semantic properties, in order to simplify the theorem proving effort. A more complete system would define an implementation function that corresponds to each predicate, and then prove that the function implies the semantic property.

For some predicates we define the negated predicate seperately from the predicate. For example the `notdef` definition:

abbreviation *notdef* :: *vname* ⇒ *J-prog* ⇒ *expr* ⇒ *heap* ⇒ *locals* ⇒ *expr* ⇒ *heap* ⇒ *locals* ⇒ *bool* **where**
notdef var prog e h l e′ h′ l′ ≡ *prog* ⊢ ⟨*e*,(*h*,*l*)⟩ → ⟨*e′*,(*h′*,*l′*)⟩ & *l var* = *l′ var*

Here the semantic property implied by the definition is that as expression `e` for a given heap `h` and local variables `l` evaluates to `e'` in (*h′*, *l′*) the local variable `var` refers the same value within the local variable environment before the evaluation and after. It additionally enforces that these expressions do indeed evaluate from `e` to `e'`.

4.4 Temporal Operators

Temporal operators within TRANS all correspond to functions within Isabelle/HOL. These give a satisfying decision for a list of expression, state pairs. This relates to

a possible path of evaluation for the expression to take. By using recursion over lists, Isabelle's standard induction tactics provide much help for proving properties about the temporal operators, which fits our shallow embedding approach. Additionally we can use HOL's Exists and Forall quantification, and associated lemmas for reasoning about the exists and forall quanitification over paths in CTL.

4.5 Actions

For each TRANS language primitive there is a corresponding function, that performs the general action. An example of the replacement function is given in Fig 3. The The `replace` function recurses over the structure of Jinja expressions, replacing any expression or sub-expression within its first argument that is equal to its second argument with its third argument.

4.6 Constant Propagation

The proof of soundness for constant propagation is structured as follows:

1. Prove that in the out state of `v := c` that the variable `v` holds the value of the constant `c`
2. Prove that if at the beginning state of a path variable `v` holds the value `c` and `v` isn't redefined then it will still hold that value at the end of the path.
3. Prove that if for all paths entering an assignment `x := v` that if `v` has the value `c` at the in state, and `x` is assigned to `v` that it results in the variable `x` holding the value `c`, in other words that `x := c` evaluates to the same state as `x := v` when `v` equals `c`

We now sketch some details of how this is performed within Isabelle. Note that Within the following section, extracted from the Isabelle script, `v` is referred to as *sub-var*, `x` as *ass-var* ahd `c` as *const*.

init abbreviates the initial conditions of the until operator, in other words, that *sub-var* is assigned a constant.

abbreviation *init* :: *vname* ⇒ *val* ⇒ *J-prog* ⇒ *expr* ⇒ *state* ⇒ *expr* ⇒ *state* ⇒ *bool*
where
init sub-var const prog e s e′ s′ ≡ (*e* = *sub-var* := (*Val const*)) & *prog* ⊢ ⟨*e*,*s*⟩ → ⟨*e′*,*s′*⟩

It is then trivial to show that *sub-var* holds the value of *const* within the successor state, which corresponds to step (1).

lemma *init sub-var const prog e* (*h*,*l*) *e′* (*h′*, *l′*) ==> *l′ sub-var* = (*Some const*)

The condition $\overleftarrow{A}(\neg def(v)\ U\ stmt(v := c))$ corresponds to this function. Importantly, being just a standard Isabelle/HOL function we can use standard inductive tactics to reason about the until's predicates set of states.

fun *side-cond* :: *vname* ⇒ *val* ⇒ *J-prog* ⇒ *expr* ⇒ *state* ⇒ (*expr* ∗ *state*) *list* ⇒ *bool* **where**

side-cond sub-var const prog e s [(*e′*,*h′*,*l′*)] = *init sub-var const prog e s e′* (*h′*,*l′*) |
side-cond sub-var const prog pe ps ((*e*,*h*,*l*)#(*e′*,*h′*,*l′*)#*es*) = (*notdef sub-var prog e h l e′ h′ l′* & *side-cond sub-var const prog pe ps* ((*e′*,(*h′*,*l′*))#*es*))

We now show, that the state at the final node in the side condition chain, is the required state, ie step (2) of the proof.

lemma [| *side-cond sub-var const prog e s es*; *es* ≠ [] ; (*e*,*h*,*l*) = (*last es*) |] ==> *l sub-var* = (*Some const*)

Formulating step (3) can then be done as follows. The assumption states that *sub-var* is equal to `const` in the the initial state.

lemma [|*lcl s sub-var* = *Some const* ; *s* = (*h*,*l*)|] ==> (*P* ⊢ ⟨*ass-var*:=(*Var sub-var*), *s*⟩ →∗ ⟨*unit*, (*h*,*l*(*ass-var*↦*const*))⟩)

5 Conclusions and Future Work

TRANS as a domain specific language for compiler optimisations has been shown to be expressive, and somewhat efficiently implementable. Foundations have been laid for its formal analysis, starting with proofs of soundness for simple optimisations. There is much work yet to be done. Some elements of the formalisation are incomplete, such as the embedding of all of the remaining components of the TRANS language. More optimisations need to proved sound, including the list cited in Section 2.2. Finally implementing a system for applying the optimisations as a phase for the `Jinja` compiler within Isabelle/HOL would complement the existing work on `Java` that was based on the Soot system.

fun *replace* :: *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr* **where**
replace init from to = (*if* (*from* = *init*) *then to else init*) |
replace (*Cast cls e*) *from to* = (*Cast cls* (*replace e from to*)) |
replace (*l*≪*bop*≫*r*) *from to* = ((*replace l from to*)≪*bop*≫(*replace r from to*)) |
replace (*V*:=*e*) *from to* = (*V* := (*replace e from to*)) |
replace (*e*·*V*{*C*}) *from to* = ((*replace e from to*)·*V*{*C*}) |
replace (*x*·*V*{*C*} := *e*) *from to* = ((*replace x from to*)·*V*{*C*} := (*replace e from to*)) |
replace (*e*·*meth′*(*args′*)) *from to* = ((*replace e from to*)·*meth′*(*args′*)) |
replace {*a*:*t*;*e*} *from to* = {*a*:*t*;(*replace e from to*)} |
replace (*e1*;;*e2*) *from to* = ((*replace e1 from to*);;(*replace e2 from to*)) |
replace (*Cond c y n*) *from to* = (*Cond* (*replace c from to*) (*replace y from to*) (*replace n from to*)) |
replace (*while* (*c*) *b*) *from to* = (*while* (*replace c from to*) (*replace b from to*)) |
replace (*throw e*) *from to* = (*throw* (*replace e from to*)) |
replace (*TryCatch e c n ce*) *from to* = (*TryCatch* (*replace e from to*) *c n* (*replace ce from to*))

Fig. 3. Replacement function

References

1. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education;, 2nd edition, 2007.
2. A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.
3. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.
4. S. Kalvala, R. Warburton, and D. Lacey. Program transformations using temporal logic side conditions. Technical Report 439, Department of Computer Science, University of Warwick, 2008.
5. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
6. T. Nipkow. Jinja: Towards a comprehensive formal semantics for a Java-like language. In H. Schwichtenberg and K. Spies, editors, *Proof Technology and Computation*, pages 247–277. IOS Press, 2006.
7. A. Pnueli and G. Zaks. Translation validation of interprocedural optimizations. In *Proceedings of the 4th International Workshop on Software Verification and Validation (SVV 2006)*. Computing Research Repository (CoRR), Aug. 2006.
8. R. Warburton and S. Kalvala. From specification to optimisation: An architecture for optimisation of java bytecode. In *CC*, pages 17–31, 2009.

Author Index