Proof-Carrying Plans

Christopher Schwaab[1], Ekaterina Komendantskaya[2], Alasdair Hill[2], František Farka[1,2], Ronald P. A. Petrick[2], Joe Wells[2], and Kevin Hammond[1]

[1] School of Computer Science, University of St Andrews, UK
{cjs26,ff32,kh8}@st-andrews.ac.uk
[2] Department of Computer Science, Heriot-Watt University, UK
{ek19,ath7,rpp6,jbw}@hw.ac.uk

Abstract. It is becoming increasingly important to verify safety and security of AI applications. While declarative languages (of the kind found in automated planners and model checkers) are traditionally used for verifying AI systems, a big challenge is to design methods that generate verified executable programs. A good example of such a *"verification to implementation"* cycle is given by automated planning languages like PDDL, where plans are found via a model search in a declarative language, but then interpreted or compiled into executable code in an imperative language. In this paper, we show that this method can itself be verified. We present a formal framework and a prototype Agda implementation that represent PDDL plans as executable functions that inhabit types that are given by formulae describing planning problems. By exploiting the well-known Curry-Howard correspondence, type-checking then automatically ensures that the generated program corresponds precisely to the specification of the planning problem.

Keywords: AI Planning · Curry-Howard Correspondence · Constructive Logic · Verification · Dependent Types.

1 Motivation

Declarative programming languages have long provided convenient formalisms for knowledge representation and reasoning, ranging from Lisp and Prolog in the 1960s-1980s to modern SMT solvers [3,2], model checkers [13], and automated planners [4,11]. Common features of such languages typically include a clear logic-based syntax, a well-understood declarative semantics, and an inference engine that produces sound results with respect to the semantics.

As AI applications become increasingly deployed in the real world, e.g. in self-driving vehicles or autonomous robots, so safety and security issues are becoming increasingly important. Existing ad-hoc software development approaches do not provide the strong confidence levels that the public expects from such applications. It is tempting to envisage that declarative languages will play an increasingly important role in verifying the safety and security of real-world AI applications. Ideally, such languages could become vehicles for *proof-carrying code*, an approach in which all relevant verification properties are directly embedded in the source code [12]. To make this possible, we must supplement the ability to prove that a property A holds in a theory T (denoted

$T \vdash A$) with robust mechanisms that can generate a program p that executes according to the specification A, together with a proof that p satisfies A (denoted $T \vdash p : A$). Such an approach would embed verification directly as an integral part of the implementation cycle for AI applications.

The well-known *Curry-Howard correspondence* [18,19], tells us, of course, that some proofs in intuitionistic first-order logic can be represented as computable functions. In this case, first-order formulae are seen as *types*, and proofs are seen as terms that inhabit those types. For example, when we write $T \vdash p : A$, we mean that a *proof term* p is an *executable program* that satisfies proposition A, and moreover that this inference is sound, i.e. that $T \vdash p : A$ holds in some formal system.

Until recently, the significance of the Curry-Howard correspondence has been predominantly theoretical. The main impediment to its practical application has been the immaturity of programming languages that could fully implement this idea. For example, in order to express verification properties of AI applications, a language that could infer $T \vdash p : A$ must possess at least first-order types. Moreover, it should ideally also possess *dependent types*. Several dependently-typed languages have now become available and increasingly practical, e.g., Coq, Agda, and Idris. This development has made it possible to re-open the discussion of the actual practical value of the *Curry-Howard correspondence*. For example, in [6,7] Fu et al. have given a Curry-Howard interpretation for first-order Horn clauses and the resolution algorithm; and Urzyczyn and Schubert [16] have given a constructive semantics for answer set programming.

In this paper, we turn our attention to AI Planning languages [4,11] – a rapidly growing research and engineering area that develops methods and tools for generating plans from declarative problem specifications. We show that the Planning Domain Definition Language (PDDL) [11] is a natural domain for the Curry-Howard implementation of declarative reasoning. In particular, specifications of planning problems that are usually written in first-order logic can be expressed naturally as types, and executable plans that are generated by PDDL can be formalised as programs that inhabit those types. Type checking thus verifies that correct executable programs are generated from specifications *via* the automated planning tool. We provide a proof-of-concept implementation [17] in the dependently-typed language Agda.

2 Example: Proof-Carrying PDDL

Figure 1 shows a snippet of PDDL code that describes the classic Blocksworld domain, a simple planning task for a robot assembling a tower from bricks. It defines a set of predicates (`handEmpty`, `holding`, `onTable`, `on`, `clear`) and an action `pickup_from_table` that must satisfy certain pre- and post-conditions ("effects") that are expressed using those predicates. Several such actions are usually defined as part of a planning domain. In addition, a grounded problem will also be supplied to the planner, e.g., to form a stack of blocks `a` on `b` on `c`, given that `a`, `b` and `c` are initially on the table (but not on each other). Given the domain and problem definitions, an automated planner will initiate an algorithm (e.g., a search procedure) to generate a sequence of actions that satisfy the specification and the goal. In our case, one possible solution is:

```
Plan1 = pickup_from_table b; putdown_on_stack b c;
  pickup_from_table a; putdown_on_stack a b.
```

Definition of a Planning Domain:

```
(define (domain blocksworld)
    (:requirements :strips)
     (:predicates
         (handEmpty)
         (holding ?x)
         (onTable ?x)
         (on ?x ?y)
         (clear ?x) )

  (:action pickup_from_table
      :parameters
        (?x)
      :precondition
        (and (handEmpty)
             (onTable ?x)
             (clear ?x) )
      :effect
     (and (not (handEmpty))
         (not (onTable ?x))
           (holding ?x) )))
```

Definition of a Planning Problem:

```
(define (problem blocksworld1)
  (:domain blocksworld)
  (:objects a b c)
  (:init (onTable a)
         (onTable b)
         (onTable c)
         (clear a)
         (clear b)
         (clear c)
         (handEmpty) )
  (:goal (and (on a b) (on b c))))
```

Fig. 1. *The Blocksworld: a code snippet defining the planning domain and a planning problem.*

We would like to have an implementation of this planning language where an executable function $plan_1$ is generated from the planning domain and problem, such that $plan_1$ corresponds to the actions of `Plan1` and has a type `onTable a` $\wedge$ `onTable b` $\wedge$ `onTable c` $\wedge$ `clear a` $\wedge$ `clear b` $\wedge$ `clear c` $\wedge$ `handEmpty` $\rightsquigarrow$ `on a b` $\wedge$ `on b c`. If this judgement type-checks, then we will obtain a verified program $plan_1$ that can be later compiled and executed. As we will show in the rest of the paper, this task is far from trivial. Although the Curry-Howard correspondence tells us that, in principle, (intuitionistic) first-order proofs have a computational meaning, it is not enough for us to just formulate arbitrary proofs. Firstly, we need to formulate a generic and automatable approach to translate PDDL domains and problems into the dependently-typed setting. In addition, we need to devise our calculus in such a way as to ensure that the programs that inhabit the types give us the actual executable plans in the PDDL sense. In this paper, we therefore develop two parallel narratives. The first sets up the general method in mathematical notation independently of the concrete implementation. The second illustrates the important engineering aspect of this work, with reference to the intricacies of the Agda encoding that we give in [17]. The two parallel story lines merge when we come to the main result of this paper: the formal proof of soundness of the proof terms that implement the plans. We state this in standard mathematical notation, but delegate the proof checking to Agda. In time, we envisage that our Agda prototype will become a fully fledged program for generation of executable code from planners, while maintaining the guarantees of soundness of the generated code relative to the plan specification.

$$
\begin{array}{rrl}
\text{Atomic Formulae} & At \ni A & ::= R\ C_1 \ldots C_n \\
\text{Formulae} & Form \ni P, Q & ::= A \mid \neg A \mid P \wedge Q \\
\text{Polarities} & \mathcal{P}olarity \ni t & ::= + \mid - \\
\text{States} & \mathcal{N} \ni M, N & ::= [\,] \mid [A^t, N] \\
\text{Plans} & \mathcal{P}lan \ni f & ::= \text{halt} \mid \alpha; f \\
\text{Contexts} & \Gamma \ni \Gamma_1, \Gamma_2 & ::= \alpha : M \rightsquigarrow N
\end{array}
$$

Fig. 2. *Definitions of Formulae, States, Plans and Actions, given a set of predicates* $\mathcal{R} = \{R, R_1, R_2, ..., R_n\}$*, a set of constants* $\mathcal{C} = \{C_1, C_2, ..., C_k\}$*, and a set of constant actions* $\{\alpha, \alpha_1, \ldots, \alpha_m\}$.

3 Planning Problems as Types

In their development of the STRIPS planner, Fikes and Nilsson presented an inference system for planning languages that is based on the notion of states, or possible *worlds*. The worlds are sets of atomic formulae, that interpret complex formula of the planning domain. Operators that are defined on the worlds interpret planning actions, and rewrite the worlds by adding and deleting the atomic formulae. The inference algorithm thus starts in an initial world and ends in a goal world by the repeated application of the operators. The system is sound in the sense that the resultant world model satisfies the goal. We now show how to work with STRIPS predicates directly in a type-system, yielding proof obligations that will be fulfilled by plan execution.

3.1 Formal Language and its Declarative Semantics

We assume a finite set of predicates $\mathcal{R} = \{R, R_1, R_2, ..., R_n\}$ each R_i of fixed arity, and a finite set $\mathcal{C} = \{C_1, C_2, ..., C_k\}$ of constants (also known as "objects"). The standard definition of first-order formulae is given in Figure 2. It has two notable restrictions: the formulae do not admit variables, and only atomic formulae can be negated. The former restriction, together with the assumption that there are only finitely many constants, ensures that the set of all atomic formulae is finite, which makes it possible to take the *closed world assumption* [15], and ensure the decidability of set membership on possible worlds. The latter restriction means that disjunction is not definable in our language. In PDDL, two key restrictions apply to the use of disjunction. Firstly, all formulae are pre-compiled into disjunctive normal form. Secondly, no "actions" can have disjunctive "effects", i.e. they cannot give rise to disjunctive post-conditions. Thus, our second restriction actually adheres to the practice of PDDL plan specification and search.

Example 1. Given the syntax of Figure 2, `handEmpty` $\wedge$ $\neg$`onTable a` is a formula.

The inductive definitions of Figure 2 are given as data type definitions in our Agda implementation (Figure 3). We provide a generic Agda module, *PCPlans*, that is parametric in predicates and actions. For each planning problem, the set of predicates $\mathcal{R}$ may then be defined concretely, as in the *Blocksworld* module. Propositional equality

```
module PCPlans
  {R : Set }
  {isDE : IsDecEquivalence
          {A = R} (_≡_) }
where

data Form : Set where
  _∧_ : Form → Form → Form
  ¬_ : R → Form
  atom : R → Form
```

```
module Blocksworld where

data C : Set where
  a b c : C

data R : Set where
  handEmpty : R
  onTable clear holding : C → R
  on : C → C → R

open import PCPlans {R} {isDE}
```

Fig. 3. *Left: Module PCPlans giving a general-case Agda definition of a formula, following the set-up of Figure 2. Right: Module Blocksworld giving specification of the particular Blocksworld domain from Figure 1: listing its constants and predicates.*

```
World : Set
World = List R
```

```
data Polarity : Set where
  + − : Polarity
```

```
neg : Polarity → Polarity
neg + = −
neg − = +
```

Fig. 4. *Agda definitions of worlds as lists of atomic formulae, polarities. (Module PCPlans)*

on atomic formulae must be shown to be decidable for the particular planning problem. As we will show later, this property is needed in order to manipulate world representation. Since our implementation of the *PCPlans* module takes a generic approach, a proof that propositional equality $\_ \equiv \_$ for $\mathcal{R}$ is decidable must also be provided as a module parameter. This explains the declaration of the main module:

```
module PCPlans {R : Set } {isDE : IsDecEquivalence {A = R} (_≡_) }
```

When we instantiate $\mathcal{R}$ with a finite set of predicates for each planning problem, we need to instantiate $isDE$ with a proof that propositional equality for this particular problem is indeed decidable. In [17] we show how to automate such "boilerplate" proofs for any given $\mathcal{R}$, using reflection.

Given a set w of atomic formulae (called a *world*), a formula P *is satisfied by* w if $w \vDash_+ P$ can be derived using the rules of Figure 5. In Agda, we take advantage of the extensive library of list operations, and so define worlds as lists of atomic formulae, as shown in Figure 4. Figure 7 gives an Agda definition of the entailment relation.

Example 2. Given the world $w_1 = \{\texttt{handEmpty}\}$, $w_1 \vDash_+ \texttt{handEmpty} \wedge \neg\, \texttt{onTable a}$.

It might be expected that the rule for conjunction with negative polarity to be given by two additional rules: $\dfrac{w \vDash_- P}{w \vDash_- P \wedge Q}$ and $\dfrac{w \vDash_- Q}{w \vDash_- P \wedge Q}$. However, our current rule is sound given the syntax restrictions, and it simplifies our reasoning on decidability of normalisation, which we define next.

$$\frac{w \Vdash_t P \quad w \Vdash_t Q}{w \Vdash_t P \wedge Q} \qquad \frac{w \Vdash_{-t} A}{w \Vdash_t \neg A} \qquad \frac{A \in w}{w \Vdash_+ A} \qquad \frac{A \notin w}{w \Vdash_- A}$$

Fig. 5. *Declarative interpretation of formulae. We define $-t$ by taking $-+ = -$ and $- - = +$.*

```
data _⊩[_]_ : World → Polarity → Form → Set where
  flip : ∀{w t A} → w ⊩[ neg t ] (atom A) → w ⊩[ t ] ¬ A
  both : ∀{w t P Q} → w ⊩[ t ] P → w ⊩[ t ] Q → w ⊩[ t ] P ∧ Q
  somewhere : ∀{w a} → a ∈ w → w ⊩[ + ] atom a
  nowhere : ∀{w a} → a ∉ w → w ⊩[ − ] atom a
```

Fig. 6. *Agda definition of the entailment relation given in Figure 5. (Module PCPlans)*

3.2 Operational Semantics, States and Types

Matching the declarative-style semantics of Figure 5, we can define an operational semantics, given by a normalisation function that acts directly on formulae and computes lists of atomic formulae with polarities. A *state* is defined as a list of atomic formulae with polarities, as in Figure 2. By a small abuse of notation, we will use $\in$ to denote list membership, as well as set membership. The function $\downarrow_t$ *normalises* a formula to a state:

$$\begin{aligned}(P \wedge Q) \downarrow_t N &= Q \downarrow_t P \downarrow_t N \\ \neg A \downarrow_t N &= A \downarrow_{-t} N \\ A \downarrow_t N &= [A^t, N]\end{aligned}$$

We write $P \downarrow_t$ to mean $P \downarrow_t [\,]$.

Example 3. Continuing with the previous examples, we have:
$(\texttt{handEmpty} \wedge \neg\texttt{onTable a}) \downarrow_+ = [\texttt{handEmpty}^+, \texttt{onTable a}^-]$.

As might be expected, while the definition of the entailment relation $\Vdash_t$ is given as an inductive data type in Agda, normalisation is defined as a function (Figure 7). Note that, in order to bring the disjunction into this language in any future extensions, normalisation function for minus could be amended, to allow for non-determinism. Normalisation is sound relative to the declarative interpretation of formulae. Given a state N, define a *well-formed world* w_N to contain all A such that $A^+ \in N$ and contain no A's such that $A^- \notin N$. Generally w_N is not uniquely defined, and we use the notation $\{w_N\}$ to refer to the (necessarily finite) set of all w_N. We then have the result:

Theorem 1 (Soundness and completeness of normalisation). *Given a formula P and a world w, it holds that $w \Vdash_t P$ iff $w \in \{w_{P\downarrow_t}\}$.*

Proof. ($\Rightarrow$) is proven by induction on the derivation of $w \Vdash_t P$. ($\Leftarrow$) follows by induction on the shape of P, cf. the attached Agda file [17] for the fully formalised proof.

Example 4. If $N = (\texttt{handEmpty} \wedge \neg\texttt{onTable a}) \downarrow_+$, then w_N may be given by e.g. $w_1 = \{\texttt{handEmpty}\}$, or $w_2 = \{\texttt{handEmpty}, \texttt{onTable b}\}$, or any other world containing `handEmpty` but not `onTable a`. The given formula will be satisfied by any such w_N.

```
NPred : Set
NPred = List (Polarity × R)

_↓[_]_ : Form → Polarity → NPred → NPred
P ∧ Q ↓[ t ] N = Q ↓[ t ] P ↓[ t ] N
¬ x ↓[ t ] N = (neg t , x) :: N
atom x ↓[ t ] N = (t , x) :: N
```

Fig. 7. *Agda definition of the normalisation function. (Module PCPlans)*

```
data Plan : Set where
  doAct : Action → Plan → Plan
  halt : Plan

data Action : Set where
  pickup_from_table_b : Action
  pickup_from_table_a : Action
  putdown_on_stack_b_c : Action
  putdown_on_stack_a_b : Action

plan₁ : Plan
plan₁ = doAct pickup_from_table_b
        (doAct putdown_on_stack_b_c
        (doAct pickup_from_table_a
        (doAct putdown_on_stack_a_b
          halt)))
```

Fig. 8. *Agda abstract definition of a Plan according to Figure 2, in module PCPlans. A concrete instantiation of the set of actions, a concrete plan $plan_1$ in module Blocksworld.*

Theorem 1 will allow us to work with states at the type level, while keeping the link to the standard PDDL formula syntax and declarative semantics.

We finally define actions and plans. Given a halting state halt, and a finite set of constant *actions* $\{\alpha, \alpha_1, \ldots, \alpha_m\}$, we define *plans* inductively as sequences of action names ending with halt, cf. Figure 2. Once again, we show an instantiated version of the Agda definition of actions in Figure 8, with actions specified as per the Blocksworld problem. In the Agda prototype [17], we first develop the code for an abstract set *Action*, and then instantiate it on the concrete examples. Figure 8 also shows the Agda function $plan_1$ that encodes `Plan1` given in Section 2 in PDDL syntax. Keeping in line with Section 2, a *planning domain (or a context)* Γ is a set of actions with effects, of the form $\alpha : N \rightsquigarrow M$, where α is a constant action, and N, M are states (see Figure 2). Figure 9 shows an Agda implementation of both the general definition of a context Γ and one concrete Γ_1 that corresponds to the PDDL code snippet of Figure 1.

We now move on to our main goal: to realise the Curry-Howard intuition and define a framework in which plans will inhabit normalised formulae seen as types. We wish to show that, proving that a certain (possibly composite) plan f satisfies pre- and post-conditions given by the formulae P and Q will be equivalent to typing the judgement

$$\Gamma \vdash f : P \downarrow_+ \rightsquigarrow Q \downarrow_+$$

We will say $P \downarrow_+$ is the initial state of the plan f, and $Q \downarrow_+$ is its final state. In the next section, we introduce typing rules that define derivations of these judgements.

```
Γ : Set
Γ = Action → NPred × NPred

Γ₁ : Γ
Γ₁ pickup_from_table_b =
  (atom handEmpty ∧ atom (onTable b) ∧ atom (clear b)) ↓₊ ,
  ((¬ handEmpty ∧ ¬ (onTable b) ∧ atom (holding b)) ↓₊)
Γ₁ pickup_from_table_a = ...
Γ₁ putdown_on_stack_b_c =
  (atom (holding b) ∧ atom (clear c)) ↓₊ ,
  (¬ (holding b) ∧ ¬ (clear c) ∧ atom (on b c) ∧ atom handEmpty) ↓₊
Γ₁ putdown_on_stack_a_b = ...
```

Fig. 9. *Agda definition of the concrete context* Γ_1 *in module* Blocksworld.

$$\text{NilSub}\ \frac{}{[\,] <: N} \qquad \text{ASub}\ \frac{N <: M \quad A^t \in M}{A^t, N <: M}$$

Fig. 10. Sub-typing of normalised formulae.

4 Plans as Proof Terms

4.1 Typing Rules for Planning Problems

A naïve attempt to type plans introduces two problems. First, an action $\alpha : M \rightsquigarrow N$ should not produce exactly N, but an extension of M by N. For example, picking up b from the table does not affect the fact that c is still on the table (this is known as the *STRIPS assumption* in planning [20]). To solve this problem, we introduce an override operator $M \sqcup N$:

$$M \sqcup [\,] = M$$
$$M \sqcup [A^t, N] = [A^t, M \backslash \{A^+, A^-\}] \sqcup N$$

The second problem involves applying $\alpha : M \rightsquigarrow N$ in a state M' that is stronger (has more atomic formulae) than M. For example, if b is known to be on the table, knowing that c is also on the table should not preclude picking up b. This state-weakening action corresponds to sub-typing $M <: M'$ defined in Figure 10. When we write $M <: M'$, we will say M' is a *sub-type* of M. This agrees with the usual convention that a sub-type is given by a stronger predicate. The rules of Figure 11 define how a program $f : M \rightsquigarrow N$ can be typed given some planning domain Γ. A well-typed plan $\Gamma \vdash f : M \rightsquigarrow N$ "transports" an initial state M to a goal state N. The Agda code implements the typing relation as an inductive data type with two constructors, *halt* and *seq*, following verbatim Figure 11 (see also the accompanying Agda file). To exemplify these rules, we refer again to the Blocksworld problem with the pre-condition $P_0 =$ `onTable a∧onTable b∧onTable c∧clear a∧clear b∧clear c∧handEmpty` and the post-condition $Q_0 =$ `on a b∧on b c`. Suppose that the PDDL planner proposes `Plan1`, as

$$\text{Halt}\ \frac{M' <: M}{\Gamma \vdash \text{halt} : M \rightsquigarrow M'} \qquad \text{Seq}\ \frac{M_1' <: M_1 \quad (\alpha : M_1' \rightsquigarrow M_2) \in \Gamma \quad \Gamma \vdash f : M_1 \sqcup M_2 \rightsquigarrow M_3}{\Gamma \vdash \alpha; f : M_1 \rightsquigarrow M_3}$$

Fig. 11. Well-typing relation for plans.

given in Section 2. Let $plan_1$ be the corresponding version in the precise mathematical notation of Figure 2 (cf. also its Agda version in Figure 8):

$$\begin{aligned}\text{plan}_1 = {} & \texttt{pickup\_from\_table\_b}; \texttt{putdown\_on\_stack\_b\_c};\\ & \texttt{pickup\_from\_table\_a}; \texttt{putdown\_on\_stack\_a\_b}; \text{halt}\end{aligned}$$

If $\Gamma_1 \vdash \text{plan}_1 : P_0 \downarrow_+ \rightsquigarrow Q_0 \downarrow_+$ yields a typing derivation by Figure 11, then this typing derivation verifies that plan_1 correctly implements the given planning problem in the planning domain Γ_1 (cf. also Agda code for Γ_1 in Figure 9). To make our example more readable, we will use our mathematical notation. This gives the following definition of Γ_1, corresponding to the Agda code of Figure 9:

$$\begin{aligned}\Gamma_1 = \{ & \texttt{pickup\_from\_table\_b} : \begin{array}{r}\texttt{handEmpty} \wedge \\ \texttt{onTable b} \wedge \\ \texttt{clear b}\end{array}\Big\downarrow_+ \rightsquigarrow \begin{array}{r}\neg\texttt{handEmpty} \wedge \\ \neg\texttt{(onTable b)} \wedge \\ \texttt{holding b}\end{array}\Big\downarrow_+\\ & \texttt{pickup\_from\_table\_a} : \ldots\\ & \texttt{putdown\_on\_stack\_b\_c} : \begin{array}{r}\texttt{holding b} \wedge \\ \texttt{clear c}\end{array}\Big\downarrow_+ \rightsquigarrow \begin{array}{r}\neg\texttt{(holding b)} \wedge \\ \neg\texttt{(clear c)} \wedge \\ \texttt{on b c} \wedge \\ \texttt{handEmpty}\end{array}\Big\downarrow_+\\ & \texttt{putdown\_on\_stack\_a\_b} : \ldots\}\end{aligned}$$

Let us perform the typing derivation for $\Gamma_1 \vdash \text{plan}_1 : P_0 \downarrow_+ \rightsquigarrow Q_0 \downarrow_+$. Given $P_0 \downarrow_+$, then the first action that we can apply by the Seq rule is `pickup_from_table_b`. The application of Seq demands that $P_0 \downarrow_+$ is a sub-type of the initial state of the action `pickup_from_table_b` in Γ_1. A sub-typing derivation provides such a proof, selecting the required piece of evidence from $P_0 \downarrow_+$, i.e. $\texttt{handEmpty}^+, \texttt{(onTable b)}^+, \texttt{(clear b)}^+ <: \texttt{(onTable a)}^+, \texttt{(onTable b)}^+, \texttt{(onTable c)}^+, \texttt{(clear a)}^+, \texttt{(clear b)}^+, \texttt{(clear c)}^+, \texttt{handEmpty}^+$. We have thus verified that $\text{plan}_1 = \texttt{pickup\_from\_table\_b}; f'$. To complete the proof of well-typedness and compute an action for f', we must show that the remainder of the plan is typeable. According to Seq, we now have a new state $P_1 = P_0 \downarrow_+ \sqcup \texttt{handEmpty}^-, \texttt{(onTable b)}^-, \texttt{(holding b)}^+ = \texttt{(onTable a)}^+, \texttt{(onTable b)}^-, \texttt{(onTable c)}^+, \texttt{(clear a)}^+, \texttt{(clear b)}^+, \texttt{(clear c)}^+, \texttt{(handEmpty)}^-, \texttt{(holding b)}^+$, as well as an obligation to prove $f' : P_1 \rightsquigarrow Q_0 \downarrow_+$. We can pick the next action from Γ_1: `putdown_on_stack_b_c`. Again P_1 is readily shown to be a sub-type of the pre-conditions of `putdown_on_stack_b_c`. Continuing in this way for each action in plan_1, the final state is $P_3 = \texttt{(onTable a)}^-, \texttt{(onTable b)}^-, \texttt{(onTable c)}^+,$

$P_0 : Form$
$P_0 = atom\ (onTable\ a) \wedge atom\ (onTable\ b) \wedge atom\ (onTable\ c) \wedge$
$atom\ (clear\ a) \wedge atom\ (clear\ b) \wedge atom\ (clear\ c) \wedge atom\ handEmpty$

$Q_0 : Form$
$Q_0 = atom\ (on\ a\ b) \wedge atom\ (on\ b\ c)$

$Derivation : \Gamma_1 \vdash plan_1 : (P_0 \downarrow_+) \rightsquigarrow (Q_0 \downarrow_+)$
$Derivation = ...$

Fig. 12. *Agda type-checking the derivation of* $\Gamma_1 \vdash plan_1 : P_0 \downarrow_+ \rightsquigarrow Q_0 \downarrow_+$. *We give the full code for Derivation in Appendix A or in [17].*

$(\texttt{clear a})^+, (\texttt{clear b})^-, (\texttt{clear c})^-, (\texttt{on b c})^+, \texttt{handEmpty}^+, (\texttt{holding b})^-,$ $(\texttt{on a b})^+, (\texttt{holding a})^-$. However, this is not the same state as the goal state Q_0. To resolve such cases, we have the rule Halt, eliminating all unnecessary evidence from the current state by proof of sub-typing i.e. $\Gamma_1 \vdash \text{halt} : P_3 \rightsquigarrow Q_0$. Clearly $(\texttt{on a b})^+$, $(\texttt{on b c})^+ <: P_3$ as required. We have thus verified that $\Gamma_1 \vdash \text{plan}_1 : P_0 \downarrow_+ \rightsquigarrow Q_0 \downarrow_+$. In Agda, the above derivation will amount to type-checking the function *Derivation* as shown in Figure 12. If it type-checks, then we know that $plan_1$ can be soundly executed as a function. Proving this property in general is the subject of the next section.

4.2 Computational Characterisation of Plans: Soundness of Plan Execution

The proof of $\Gamma_1 \vdash \text{plan}_1 : P_0 \downarrow_+ \rightsquigarrow Q_0 \downarrow_+$ provides evidence that the *execution* of plan_1 on a world satisfying P_0 produces a new world satisfying Q_0. Generally, the inference of $\Gamma \vdash f : M \rightsquigarrow N$, with $f = \alpha_1; \ldots; \alpha_j; \text{halt}$ corresponds to successively applying actions $\alpha_1 \ldots \alpha_j$ to states $M, M_1, \ldots M_j$ in a sequence of state transitions, satisfying $N <: M_j$. We now prove that the plan f thus inferred indeed has a computational meaning, i.e. can be *evaluated*, and that the result of its evaluation is sound. To state this, we need to define an *evaluation function* $[\![.]\!]$ that will interpret actions on worlds. Recall that every state N maps to a world w_N. Let us use notation σ for an arbitrary mapping (an *action handler*) that maps each action $\alpha : M \rightsquigarrow N$ to insertions and deletions on the world w_M according to α's action on M. We then define the evaluation function $[\![]\!]^\sigma_w$ that *evaluates* a plan to a world (according to a given world w and action handler σ):

$$[\![\text{halt}]\!]^\sigma_w = w$$
$$[\![\alpha; f]\!]^\sigma_w = [\![f]\!]^\sigma_{(\sigma\ \alpha\ w)}$$

We say that an action handler σ is *well-formed* with respect to a context Γ if, for any $w \in \{w_M\}$, $M' <: M$ and $\alpha : M' \rightsquigarrow N$ in Γ it follows that $(\sigma\ \alpha\ w) \in \{w_{M \sqcup N}\}$. Figure 13 shows Agda definitions of an action handler and evaluation action.

Canonical Handler. In order to be constructive in our further claims, and to provide a practical solution to the quest for a well-formed handler, we first define a *canonical*

```
ActionHandler : Set
ActionHandler = Action → World → World

⟦_⟧ : Plan → ActionHandler → World → World
⟦ doAct α f ⟧ σ w = ⟦ f ⟧ σ (σ α w)
⟦ halt ⟧ σ w = w

σα : NPred → World → World
σα [] w = w
σα ((+ , x) :: N) w = x :: σα N w
σα ((− , x) :: N) w = remove x (σα N w)

canonical−σ : Γ → ActionHandler
canonical−σ Γ₁ α = σα (proj₂ (Γ₁ α))
```

Fig. 13. *Agda code for an action handler, an evaluation function and a canonical handler.*

handler for a given context (planning domain). Firstly, we define a function σ_α that constructs a world from a state:

$$\begin{aligned} \sigma_\alpha\ [\,]\ w &= w \\ \sigma_\alpha\ [A^+, N']\ w &= \sigma_\alpha\ N'\ (w \cup \{A\}) \\ \sigma_\alpha\ [A^-, N']\ w &= \sigma_\alpha\ N'\ (w \setminus \{A\}) \end{aligned}$$

Next, given a context Γ, we apply σ_α to N for each $\alpha : M \rightsquigarrow N$ in Γ; thus obtaining a canonical mapping from actions and worlds into worlds, as required. The resulting canonical action handler $canonical{-}\sigma\ \Gamma$ (defined formally in Figure 13) is well-formed, as long as the states to which it is applied are consistent, in the following sense:

Implicit consistency assumption: for every state N, if $A^t \in N$ then $A^{-t} \notin N$.

Proposition 1. *Given a context* Γ*, the canonical handler* $canonical{-}\sigma\ \Gamma$ *is well-formed with respect to* Γ.

Proof. The proof starts with considering an arbitrary state M with $w \in \{w_M\}$, an arbitrary $A^t \in M$, and an arbitrary action α in Γ such that $M' <: M$ and $\alpha : M' \rightsquigarrow N$. It proceeds by considering two cases, when $t = +$ and $t = -$, and consequently when $A \in w$ or $A \notin w$. In each of these cases, it considers all possible effects of σ_α (i.e. formula deletions and insertions) in the process of constructing the world $w' = canonical{-}\sigma\ \Gamma\ \alpha\ w$. The attached Agda file gives the full proof. It uses the implicit consistency assumption to eliminate the cases when states are inconsistent and hence when more than one choice for deletion/insertion is possible.

The next two theorems show that executing a well-typed plan f by the evaluation function $[\![f]\!]^\sigma_w$ is *sound*, for any well-formed handler σ.

Theorem 2 (Soundness of evaluation for normalized formulae). *Suppose* $\Gamma \vdash f : M \rightsquigarrow N$. *Then for any* $w \in \{w_M\}$, *and any well-formed handler* σ, *it follows that* $[\![f]\!]^{\sigma}_{w} \in \{w_N\}$.

Proof. The proof proceeds by structural induction on the typing derivation $\Gamma \vdash f : M \rightsquigarrow N$.

*Case 1 (*Halt*).* By assumption $w \in \{w_M\}$ and thus because $N <: M$, it follows $w \in \{w_N\}$. Since $[\![\text{halt}]\!]^{\sigma}_{w} = w$, we get $[\![\text{halt}]\!]^{\sigma}_{w} \in \{w_N\}$ as required.

*Case 2 (*Seq*).* Note that $f = \alpha; f'$ and therefore $\alpha : M' \rightsquigarrow M_2$ is in Γ and $\Gamma \vdash f' : M \sqcup M_2 \rightsquigarrow N$ by inversion on $\Gamma \vdash \alpha; f' : M \rightsquigarrow N$. Then by induction every $w' \in \{w_{M \sqcup M_2}\}$ gives $([\![f]\!]^{\sigma}_{w'}) \in \{w_N\}$ for any well-formed σ. However, by the well-formedness of σ and because $w \in \{w_M\}$, we have $(\sigma\ \alpha\ w) \in \{w_{M \sqcup M_2}\}$. Thus $[\![f']\!]^{\sigma}_{(\sigma\ \alpha\ w)} \in \{w_N\}$ and therefore $[\![f]\!]^{\sigma}_{w} \in \{w_N\}$.

Theorem 3 (Soundness of evaluation). *Suppose* $\Gamma \vdash f : P \downarrow_{+} \rightsquigarrow Q \downarrow_{+}$ *then for any* w *such that* $w \vDash_{+} P$, *and any well-formed* σ *it follows* $[\![f]\!]^{\sigma}_{w} \vDash_{+} Q$.

Proof. By assumption $w \vDash_{+} P$ and by the completeness of normalisation (Theorem 1), we have $w \in \{w_{P\downarrow_{+}}\}$. Then from Theorem 2, we have $[\![f]\!]^{\sigma}_{w} \in \{w_{Q\downarrow_{+}}\}$. Thus by the soundness of normalisation (Theorem 1), obtain $[\![f]\!]^{\sigma}_{w} \vDash_{+} Q$.

Thus the derivation of a type for a plan f induces a proof that the execution of a plan in world w is correct. Although neither of the above theorems depends on the implicit consistency assumption for its proofs, the existence of a well-formed and canonical handler is predicated upon the consistency assumption. Our Agda implementation of a canonical handler (cf. Figure 13) allows us to fully harness the computational properties of plans. For the Blocksworld example, we can directly evaluate $[\![\text{plan}_1]\!]^{\sigma}_{w}$ by plugging in:
– in place of w – the world resulting from computing $\sigma_{\alpha}(P_0 \downarrow_{+} [\,]) \varnothing$. (To see this, recall that P_0 is the formula that described the initial state in all examples of the previous section, and $P_0 \downarrow_{+} [\,]$ is the state resulting from normalising P_0.)
– and in place of σ – the canonical handler for Γ_1. (Recall that Γ_1 is the context that defined the given planning domain in the previous section.)

In Agda, we simply evaluate the term:

$$[\![\ plan_1\ ]\!]\ (canonical{-}\sigma\ \Gamma_1)\ (\sigma\alpha\ (P_0 \downarrow[\ +\ ]\ [])\ [])$$

Evaluation of this term results, just as we manually computed in the last section, in the world $w' = \{$ `handEmpty,on a b,on b c,clear a,onTable c`$\}$ (in Agda syntax: $handEmpty :: (on\ a\ b) :: (on\ b\ c) :: (clear\ a) :: (onTable\ c) :: []$). That is, the world that corresponds to the state P_3 of the previous section. Observe that $w' = \sigma_{\alpha}\ P_3\ \varnothing$.

5 Discussion, Conclusions, and Future Work

We have given a proof of concept formalisation of a subset of PDDL plans in type theory. In line with the Curry-Howard approach to first-order logic, we formulated an inference system that treats planning domains as types, and generated plans as functions that

inhabit these types. Type-checking then ensures the soundness of these executable functions relative to the specifications given as types. This paper does not cover the whole PDDL syntax, nor does it implement the search and decision procedures of a usual automated planner e.g. the *Stanford Research Institute Problem Solver* STRIPS [4]. Rather, our contribution is in setting up the original design of a method of Curry-Howard approach to AI planning languages in general, as well as showing the feasibility of its successful implementation in a dependently typed language, such as Agda. This dual purpose has determined our style of presentation, in which the formal method has been given in parallel with, but independently from, the Agda code.

Further experiments with plans: In the accompanying implementation [17], we provide a second fully implemented example of a PDDL domain and plan checking in Agda: for a *Logistic* planning problem. The problem consists of finding the best route (airplanes, tracks, cities of call) to deliver a given parcel to a given office. The experiment showed that, once the main Agda implementation is set up, instantaiting it with various problems only takes a routine boilerplate code (such as e.g. proofs of decidability of equality on predicates). Generation of this boilerplate code can in future be fully automated using code reflection. Throughout our implementation, we have been working with plans generated by an on-line PDDL editor `http://editor.planning.domains/`. In the future, a parser can be added to convert PDDL syntax directly into Agda.

With the view to future extensions, our Agda code is designed in a modular way, as Section 3.1 illustrates: the main Agda file implementing the subset of PDDL syntax is fully generic, and its definitions are instantiated as required when a particular planning domain, such as *Blockworld* or *Logistic*, is presented. Proofs of decidability for objects and predicates for each given problem are obtained in a generic way, as well, see [17].

Computational content of plans and the implicit consistency assumption: As Proposition 1 has shown, the existence of a canonical and well-formed handler depends crucially on the implicit consistency assumption. At the same time, the proofs of Theorems 2 or 3 do not depend on the consistency assumption. Thus, as we show in Appendix B, it is possible in principle to construct planning domains and problems that violate the assumption but are accepted by the well-typedness relation of Figure 11. However, if such examples are added to the system, the implicit consistency assumption needs to be removed (or else $\bot$ will become provable, as Appendix B shows). But without the assumption, we lose the existence of a well-formed and canonical handler and thus the ability to evaluate the plans. This situation is of course illustrative of the rigour and transparency that a constructive approach brings to verification. In our case, it dictates that any practical deployment of the presented prototype needs to enforce the consistency assumption. This can be done by either embedding additional state consistency checks or by implementing states as partial functions from formulae to Booleans.

Generating executable plans from Agda: The main advantage of the presented approach is the ability to generate executable code directly from plans verified in Agda. Appendix C illustrates how first a Haskell code, and then an executable binary file, are automatically generated from the verified plan ($plan_1$ from our earlier example). Such binary files can then be directly deployed in applications such as e.g. robots. This is in contrast

to the existing practices when verified plans are separately converted into C or Python code without any guarantees of compliance of that code to the verified plans.

Related work: Verification of AI languages and applications is an active research field. In planning languages, two major trends exist. Firstly, PDDL is used to verify autonomous systems and applications, see e.g. [14]; and it has been successfully integrated within other similar languages, such as GOLOG [10], with the purpose of verifying plans written in the situation calculus [1,21]. Secondly, planning domains have been verified using model checkers [9], other automated provers such as Event-B [5], or planning support tools such as VAL [8]. The method that we have presented is complementary to these two trends. Its main difference lies in taking the perspective of *intrinsic*, rather than *external* verification. That is, the correctness of the generated plans is verified not by an external tool, such as a model checker, but is performed intrinsically within the code that implements the plans. At the same time, the code that implements the plans is inseparable from the language in which planning domains are specified. Furthermore, the executable binary files automatically extracted from Agda are not just ready for deployment, but also bear the verification guarantees provided by Agda proofs. To our knowledge, this is the first attempt to bring these benefits of the Curry-Howard approach to automated planning languages. Provisioning types for plans not only equips planners with certificates of correctness for inspection, but also provides a direct link to an implementation's type theory.

Current limitations and possible improvements: First-order planning domains and first-order types: Although the technical development of the code that we have presented takes full advantage of Agda's dependent types, the types that represent the predicates and formulae of the planning problems are given by simple types. This is because we propositionalised the planning domains. We however hope to extend this initial framework to the full first-order syntax of PDDL. This development will also involve the following extensions.

Beyond consistency assumptions; constraints: We have discussed the implicit consistency assumption that our approach imposes. More generally, we note that PDDL lacks any general method of handling consistency as well as similar but more complex constraints and invariants, such as, for example, constraints saying that `handEmpty` and `holding x` are mutually exclusive. This is a complex problem, but one for which we anticipate that our dependently-typed setting will soon provide some useful solutions.

Functions and higher-order plans: The design of this Agda prototype has revealed several limitations in state-of-the-art implementations of planning languages: e.g. their reliance on the closed word assumption and formulae grounding, the absence of functions, and the restricted use of disjunctions. Again, we see a potential of our method to overcome many of these limitations thanks to our general dependently-typed set-up, in which the use of functions, higher-order features, constraints and effect handling will be much more natural than in the current implementations.

Acknowledgements

This research has been generously supported by EPSRC Platform Grant EP/N014758/1 "The Integration and Interaction of Multiple Mathematical Reasoning Processes", EPSRC Doctoral Training Partnership, EPSRC grant EP/PP020631 "Discovery: Pattern Discovery and Program Shaping for Manycore Systems", and by EU Horizon 2020 grant ICT-779882 "TeamPlay: Time, Energy and security Analysis for Multi/Many-core heterogenous PLAtforms".

References

1. Claßen, J., Eyerich, P., Lakemeyer, G., Nebel, B.: Towards an integration of Golog and Planning. In: Veloso, M.M. (ed.) IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007. pp. 1846–1851 (2007)
2. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008)
3. Dutertre, B., De Moura, L.: The Yices SMT solver. Tech. rep. (8 2006), tool paper at http://yices.csl.sri.com/tool-paper.pdf
4. Fikes, R., Nilsson, N.J.: STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence **2**(3/4), 189–208 (1971)
5. Fourati, F., Bhiri, M.T., Robbana, R.: Verification and validation of PDDL descriptions using Event-B formal method. In: 2016 5th International Conference on Multimedia Computing and Systems (ICMCS). pp. 770–776 (Sept 2016)
6. Fu, P., Komendantskaya, E.: Operational semantics of resolution and productivity in Horn clause logic. Formal Asp. Comput. **29**(3), 453–474 (2017)
7. Fu, P., Komendantskaya, E., Schrijvers, T., Pond, A.: Proof relevant corecursive resolution. In: Kiselyov, O., King, A. (eds.) Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9613, pp. 126–143. Springer (2016)
8. Howey, R., Long, D., Fox, M.: VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In: 16th IEEE International Conference on Tools with Artificial Intelligence. pp. 294–301 (2004)
9. Khatib, L., Muscettola, N., Havelund, K.: Verification of plan models using UPPAAL. In: Rash, J.L., Rouff, C.A., Truszkowski, W., Gordon, D.F., Hinchey, M.G. (eds.) Formal Approaches to Agent-Based Systems, First International Workshop, FAABS 2000 Greenbelt, MD, USA, April 5-7, 2000, Revised Papers. Lecture Notes in Computer Science, vol. 1871, pp. 114–122. Springer (2000)
10. Levesque, H.J., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.B.: GOLOG: A logic programming language for dynamic domains. The Journal of Logic Programming **31**(1), 59–83 (1997)
11. McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., Wilkins, D.: PDDL – The Planning Domain Definition Language (Version 1. 2). Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control (1998)
12. Necula, G.C.: Proof-carrying code. In: POPL. pp. 106–119 (1997)
13. Ong, L.: Higher-order model checking: An overview. In: 30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015. pp. 1–15 (2015)

14. Raimondi, F., Pecheur, C., Brat, G.: PDVer, a tool to verify PDDL planning domains. In: ICAPS'09 Workshop on Verification and Validation of Planning and Scheduling Systems, September 20, 2009, Thessaloniki, Greece (2009)
15. Reiter, R.: On closed world data bases. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 55–76. Plenum Press, New York (1978)
16. Schubert, A., Urzyczyn, P.: Answer set programming in intuitionistic logic. Indagationes Mathematicae **29**(1), 276 – 292 (2018), l.E.J. Brouwer, fifty years later
17. Scwaab, C., Hill, A., Farka, F., Komendantskaya, E.: Proof-carrying plans: Agda implementation and examples (2018), `https://github.com/PDTypes`
18. Sorensen, M.H., Urzyczyn, P.: Lectures on the Curry-Howard Isomorphism, Studies in Logic, vol. 149 (2006)
19. Wadler, P.: Propositions as types. Commun. ACM **58**(12), 75–84 (2015)
20. Waldinger, R.J.: Achieving several goals simultaneously. Machine Intelligence **8** (1977)
21. Zarrieß, B., Claßen, J.: Decidable verification of GOLOG programs over non-local effect actions. In: Schuurmans, D., Wellman, M.P. (eds.) Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA. pp. 1109–1115. AAAI Press (2016)

A The full code for Derivation of Figure 12

```
P₀ : Form
P₀ = atom (onTable a) ∧ atom (onTable b) ∧ atom (onTable c) ∧
  atom (clear a) ∧ atom (clear b) ∧ atom (clear c) ∧ atom handEmpty

Q₀ : Form
Q₀ = atom (on a b) ∧ atom (on b c)

{- Γ₁ ⊢ plan₁ : P₀ ↓₊ ⇝ Q₀ ↓₊ -}
Derivation : Γ₁ ⊢ plan₁ : (P₀ ↓₊) ⇝ (Q₀ ↓₊)
Derivation =
  seq (atom<: (there (there (here refl)))
      (atom<: (there (there (there (there (there (here refl))))))
      (atom<: (here refl)
      ([]<: ((+ , handEmpty) ::
              (+ , clear c) ::
              (+ , clear b) ::
              (+ , clear a) ::
              (+ , onTable c) :: (+ , onTable b) :: (+ , onTable a) :: [])))))
      refl
  (seq (atom<: (there (there (there (here refl)))))
       (atom<: (here refl)
       ([]<: ((+ , holding b) ::
               ( , onTable b) ::
               ( , handEmpty) ::
               (+ , clear c) ::
               (+ , clear b) ::
               (+ , clear a) :: (+ , onTable c) :: (+ , onTable a) :: []))))
       refl
  (seq (atom<: (there (there (there (there (there (there (here refl))))))))
       (atom<: (there (there (there (there (there (there (there (there (here refl))))))))))
       (atom<: (here refl)
       ([]<: ((+ , handEmpty) ::
               (+ , on b c) ::
               ( , clear c) ::
               ( , holding b) ::
               ( , onTable b) ::
               (+ , clear b) ::
               (+ , clear a) :: (+ , onTable c) :: (+ , onTable a) :: [])))))
       refl
  (seq (atom<: (there (there (there (there (there (there (there (here refl))))))))))
       (atom<: (here refl)
       ([]<: ((+ , holding a) ::
               (− , onTable a) ::
```

```
                (− , handEmpty) ::
                (+ , on b c) ::
                (− , clear c) ::
                (− , holding b) ::
                (− , onTable b) ::
                (+ , clear b) :: (+ , clear a) :: (+ , onTable c) :: [])))
        refl
    (halt (atom<: (there (there (there (there (there (here refl))))))
          (atom<: (there (here refl))
          ([]<: ((+ , handEmpty) ::
                  (+ , on a b) ::
                  (− , clear b) ::
                  (− , holding a) ::
                  (− , onTable a) ::
                  (+ , on b c) ::
                  (− , clear c) ::
                  (− , holding b) ::
                  (− , onTable b) :: (+ , clear a) :: (+ , onTable c) :: [])))))))
```

B Experiment with Inconsistent States

```
-------------------------------------------------------
-- A simple example that demonstrates violation of the
-- implicit consistency assumpion
module PCPlans_naughty where
data R : Set where
  handEmpty : R

-- There is only one, naughty, action, which violates
-- the implicit consistency assumption
data Action : Set where
  naughty : Action

open import PCPlans {Action} {R} {isDecidable}

-- The naughty action does not have any preconditions and
-- introduces an atomic predicate and its negation as
-- postconditions
Γ1 : Γ
Γ1 naughty = ( [] , (( , handEmpty) :: (+ , handEmpty) :: []))

-- Despite the obvious inconsistency,
-- the following plan has a derivation that type checks:
plan2 : Plan
plan2 = doAct naughty (halt)

Q : Form
Q = atom (handEmpty) ∧ ¬ handEmpty

Derivation2 : Γ1 ⊢ plan2 : [] ⇝ (Q ↓[ + ] [])
Derivation2 = seq ([]<: []) refl (halt (atom<: (here refl)
  (atom<: (there (here refl))
    ([]<: ((− , handEmpty) :: (+ , handEmpty) :: [])))))

-- But, at the same time, action naughty
-- invalidates consistency of entire development
-- (given the implicit consistency assumption):
prop−inconsistent : ⊥
prop−inconsistent =
  implicit−consistency−assumption + handEmpty (proj2 (Γ1 naughty))
    (there (here refl))
    (here refl)
```

C Executable Code Extraction

In Section 4.2 we showed how to use the canonical handler to evaluate a plan. The dependently typed approach, however, offers more than just evaluation in Agda. It offers a possibility to automatically extract executable functions from Agda code. For example, we can turn the function

```
world−eval : World
world−eval = ⟦ plan1 ⟧ (canonical−σ Γ1) wP1
```

corresponding to the examples of Section 4.2 into an executable Haskell program by compiling our Agda code to Haskell using Agda's builtin extraction tools. The automatically generated Haskell code for *world−eval* is:

```
name422 = "PCPlans_blocksworld.world−eval"
d422 :: [T10]
d422 = coe
 (MAlonzo.Code.PCPlans.du1984
  (coe d410)
  (coe( MAlonzo.Code.PCPlans.du2176 (coe d210) (coe d412)))
  (coe d420))
```

To do this, the file run.agda has been created (see [17]) with a main function that outputs the result of the *world−eval* function as a string. The resulting binary file called *run* returns the following world state after executing the plan:

```
handEmpty on a b on b c onTable c clear a
```

It takes only 0.004*s* to run. The binary file can in principle be run on any independent platform, e.g. on a robot.

Generally, as the reader can find in [17], the extraction of executable code from Agda proceeds by compilation of all necessary Agda modules for run.agda, and creates a stand alone repository of Haskell code that includes all necessary Haskell functions corresponding to the Agda development in question.