# Isabelle/Spartan—A Dependent Type Theory Framework for Isabelle

Joshua Chen
University of Innsbruck, Austria

**Abstract**

This paper introduces Isabelle/Spartan, an implementation of intensional dependent type theory with cumulative universes as an object logic in the Isabelle proof assistant. In contrast to other systems supporting dependent type theory, Isabelle is based on simple type theory—yet I show how its existing logical framework infrastructure is able to handle automation tasks that are typically implemented on the source code level of dependently-typed systems. I also go some way in integrating the propositions-as-types paradigm with the declarative Isar proof language. Isabelle/Spartan supports book HoTT and the univalence axiom, and its capabilities are demonstrated by the formalization of foundational results from the Homotopy Type Theory book.

**2012 ACM Subject Classification** Theory of computation → Logic and verification; Theory of computation → Type theory; Theory of computation → Higher order logic

**Keywords and phrases** Proof assistants, Logical frameworks, Dependent type theory, Homotopy type theory

**Category** System description

**Funding** Partially supported by ERC project "SMART", starting grant no. 714034.

## 1 Introduction

Proof assistants based on dependent type theory have historically been built "from the ground up" to support their single logical foundation. In contrast to such systems are logical frameworks [11], which are designed to allow a user to work with a wide range of different object logics as environments for formalization and proof, at the potential cost of having less specific automation for any particular formalism. More recently, there has been work towards creating new logical frameworks explicitly designed to support dependent type theories as object logics [1, 5]. All of these systems are themselves dependently-typed.

In contrast, Isabelle [14, 16] is a simply-typed proof assistant and logical framework. Of its multiple object logics Isabelle/HOL is arguably the most well-known, however many other logics have been created since Isabelle's inception and are still bundled along with its distribution [9]. Among these is an early logic by Paulson [8] based on extensional Martin-Löf type theory, which has not been further developed. In light of considerable recent progress in the field, it seems appropriate to revive support for dependent type theory in Isabelle.

### Motivation

The potential benefits of support for dependent type theory in Isabelle to both the proof assistant and type theory communities seem attractive. One such benefit is the possibility of encoding other versions of dependent type theory, enabling one to rapidly experiment with different formulations and prove meta-theoretic results about them. Support for dependent type theory in a simply-typed setting will also pave the way for greater compatibility between proof assistant libraries, allowing for the porting of results between systems of different formalisms. More ambitiously, the preliminary work presented here in translating between a dependent type theoretic formalism and the Isar language suggests the possibility (momentarily ignoring concerns of consistency) of HOL-based and dependently-typed "sub-logics" coexisting under one object logic, allowing the user to choose whichever formulation best suits their particular development.

## Contributions

In this paper I introduce *Isabelle/Spartan* (Spartan), a new object logic based on dependent type theory[1].

In the first half of this paper I present an encoding of intensional dependent type theory with cumulative Russell-style universes in the Isabelle/Pure (Pure) meta-logic, and discuss issues and design decisions that arise. Due to small but significant differences between the semantics of Pure and the formalism of Martin-Löf-style type theories, a naïve translation of the latter into the former preserves neither adequacy nor soundness. The rules of Spartan have been formulated to avoid the most obvious sources of this failure; however, as of this writing, soundness and completeness of the encoding with respect to its intended semantics have not yet been proved.

In the second half, I give an overview of the implementation of the system, showing how the existing Isabelle infrastructure may be used to solve tasks—such as typechecking and term elaboration—that are typically considered specific to dependently-typed systems and implemented as routines in their source code. Progress is also made in integrating the propositions-as-types paradigm with the declarative Isar proof language [17]. Although the system currently lacks some automation, one can already easily state and prove nontrivial results from the Homotopy Type Theory book [13] in a style mostly familiar to users of dependently-typed proof assistants.

## Source code

The work presented in this paper has been implemented as a library of standard ML and Isabelle theory files. References to specific files are given as footnotes throughout. The source code is available at `https://github.com/jaycech3n/Isabelle-Spartan`.

## Related work

One of the earliest object logics for Isabelle was Paulson's Isabelle/CTT (CTT) [8] for computational type theory, in the footsteps of which the present work follows. Indeed, the ideas of implementing typechecking as a tactic and of tackling subgoals in order of decreasing "rigidity" already appear in CTT. On the other hand, as CTT implements extensional Martin-Löf type theory without universes it requires less automation for equality reasoning and universe levels than Spartan, and is also unsuited to the homotopical viewpoint.

Andromeda [5] is a new LCF-style proof assistant explicitly designed to support the formulation and subsequent use of user-defined type theories. It provides a general-purpose meta-language to define constructors, types and inference rules, as well as a small trusted nucleus that checks well-typedness and derivability of statements. It is, however, more restrictive than Isabelle in the kinds of type theory it is able to specify as it only allows the formulation of inference rules using the four judgment types of traditional Martin-Löf type theory. In particular, the interval object of cubical type theory is not amenable to this

[1] The name is inspired by talks given by Bauer [4] on minimal type theories that support univalence and the homotopical interpretation.

$$\dfrac{\begin{array}{c}[P]\\ \vdots \\ Q\end{array}}{P \Longrightarrow Q}\Longrightarrow_I \qquad \dfrac{P \Longrightarrow Q \quad P}{Q}\Longrightarrow_E \qquad \dfrac{P}{\bigwedge x.\ P}\wedge_I \qquad \dfrac{\bigwedge x.\ P(x)}{P(t)}\wedge_E$$

$$\dfrac{}{a \equiv a}\equiv_{\text{refl}} \qquad \dfrac{a \equiv b}{b \equiv a}\equiv_{\text{sym}} \qquad \dfrac{a \equiv b \quad b \equiv c}{a \equiv c}\equiv_{\text{trans}}$$

Side conditions:
- In $\wedge_I$, the variable $x$ is fresh.
- In $\wedge_E$, $t$ is any term of appropriate type.

**Figure 1** Inference rules of Isabelle/Pure.

framework. In this respect the Isabelle-based work presented here is more flexible, though one would need to consider issues of adequacy, soundness and completeness of the encodings.

Dedukti [1, 3] is a logical framework based on the λΠ-calculus Modulo, which is also able to encode a wide variety of logics including dependent type theory. It is, however, designed as a proof checker and is hence not geared towards the interactive proof assistant style of working.

Finally, one cannot speak of logical frameworks without mentioning the LF family of languages, in particular the ELF and TWELF [10, 12] systems. These differ from the work here in that they are focused more on enabling one to specify and prove properties *about* object logics instead of letting one work *inside* them, while the vision for Spartan is to allow one to do both within a single system.

## 2 The Isabelle Logical Framework

To provide context for the presentation of Isabelle/Spartan, this section briefly introduces the functionality of the Isabelle logical framework as well as the Isabelle/Pure meta-logic. Pure is a minimal logic designed primarily to enable users to encode and work with the terms, formulas and inference rules of object logics in a natural deduction style. It is based on rank-one polymorphic simple type theory with a base type *prop* to represent logical propositions, together with three constants

$$\Longrightarrow :: \mathit{prop} \to \mathit{prop} \to \mathit{prop}, \qquad \bigwedge :: (\alpha \to \mathit{prop}) \to \mathit{prop}, \qquad \equiv :: \alpha \to \alpha \to \mathit{prop}$$

expressing implication, universal quantification and equality. As is usual, let us write the Church-style universal quantifier $\bigwedge(\lambda x.\, P)$ as $\bigwedge x.\, P$. The rules governing the logical constants are shown in Figure 1. Derivability, discharge of assumptions, and side condition checking are handled by the Isabelle system itself. As a simple type theory, Pure also has lambda terms built in, enabling us to encode terms, types and judgments using higher-order abstract syntax. Alpha-conversion as well as beta- and eta-reduction is automatically handled by the system, and the Isabelle simplifier proves substitution rules for terms. In addition, the framework provides functionality to declare new base types and constants, and to state axioms and inference rules. More details can be found in the Isabelle documentation.

## 3 The Logic of Isabelle/Spartan

This section details the basic setup of the Isabelle/Spartan object logic.[2]

### 3.1 Semantics

The intended semantics of Spartan is a minimal dependent type theory consisting of the $\Pi$, $\Sigma$ and identity types, along with a countable cumulative hierarchy of Russell-style universe types. This theory itself is standard, and mostly follows the development given in the appendix of the Homotopy Type Theory book [13]. Work is ongoing to establish adequacy, soundness and completeness theorems for the encoding presented here, but until then—as noted by Paulson [8]—the object logic is best justified by directly considering the meaning explanations of the rules.

### 3.2 Judgments

We begin by declaring a meta-type $o$ for the class of terms and types of the object logic. We then declare a constructor `has_type` $:: o \to o \to prop$, written as infix (:), to encode the typing judgment. Since we have chosen to work with Russell-style universes, types are themselves terms and must have the same meta-type. Working with Tarski-style universes would allow us to maintain the type/term distinction and formulate the typing judgment constant as `has_type` $:: i \to t \to prop$, at the cost of having to introduce interpretation operators everywhere. Judgmental equality is shallowly embedded via the built-in Pure equality ($\equiv$), which forgets type information but allows us to easily reuse the Isabelle simplifier to compute terms.

Here there is a subtle but important difference between the theory and its implementation. In theory, all judgments $\Gamma \vdash t \colon T$ are entailed by an explicit context of typings, which automatically ensures that all statements only contain well-typed terms. In contrast, Spartan's variable contexts are encoded as implications in the Pure logic, which means that it is possible to form formulas $t \colon T$ containing untypable terms. However, these formulas will not (pending soundness) be provable.

### 3.3 Universes

To implement universe types we first axiomatize a hierarchy of levels isomorphic to the standard natural numbers with their usual order. We declare a meta-type $lvl$ and the constants `0`, `S` and $<$ for the zero level, successor level and the order relation. Universes are then formed by a single constructor `U` $:: lvl \to o$. Figure 2 shows the rules governing levels and universes.

### 3.4 Types and Terms

The constants for small types, their constructors and their eliminators are formulated using Church-style semantics and listed in Figure 3. Type families as well as function arguments to dependent eliminators are encoded using meta- instead of object-lambda terms. For example, in theoretical presentations the $\Sigma$-eliminator is given by a term

$$\texttt{SigInd}(A, B, C, f, p)$$

[2] Source code: `Spartan.thy`.

```
axiomatization
             0 ::  lvl
             S ::  lvl → lvl
            lt ::  lvl → lvl → prop                    (infix <)
             U ::  lvl → o
         where
        O_min:  0 < S(i)
         lt_S:  i < S(i)
     lt_trans:  i < j ⟹ j < k ⟹ i < k
  U_hierarchy:  i < j ⟹ U(i) : U(j)
 U_cumulative:  A : U(i) ⟹ i < j ⟹ A : U(j)
```

**Figure 2** Universe types.

```
axiomatization
           Pi ::  o → (o → o) → o
          lam ::  o → (o → o) → o
          app ::  o → o → o                              (infix `)

          Sig ::  o → (o → o) → o
         pair ::  o → o → o                              (<_, _>)
       SigInd ::  [o, o → o, o → o, o → o → o, o] → o

           Id ::  [o, o, o] → o
         refl ::  o → o
        IdInd ::  [o, [o, o, o] → o, o → o, o, o, o] → o
```

**Figure 3** Type and term constructors.

whose third and fourth arguments are meant to be, respectively, a type family $C\colon \big(\sum_{x\colon A} B(x)\big) \to U_i$ and a function $f\colon \prod_{x\colon A,\ y\colon B(x)} C(x,y)$ inductively defining the value of $C$ on all $p\colon \sum_{x\colon A} B(x)$. In the implementation, these have to be given as the simply-typed meta-functions $C :: o \to o$ and $f :: o \to o \to o$. However, after the $\Pi$ type has been axiomatized Isabelle's coercive subtyping functionality (Section 12.3 of [15]) is used to coerce object functions into meta functions, which allows users to ignore this distinction most of the time.

Isabelle syntax translations convert between the notations $\prod x\colon A.\, B$, $\sum x\colon A.\, B$, $\lambda x\colon A.\, b$ and $x =_A y$ to the internal representations $\texttt{Pi}(A, \lambda x.\, B)$, $\texttt{Sig}(A, \lambda x.\, B)$, $\texttt{lam}(A, \lambda x.\, b)$ and $\texttt{Id}(A, x, y)$. The types $\prod x\colon A.\, B$ and $\sum x\colon A.\, B$ are abbreviated to $A \to B$ and $A \times B$ when $B$ is a constant type family.

## 3.5 Inference Rules

Following [7], we define a translation **enc** from the judgments of dependent type theory into the Pure logic, by sending

$$x_1\colon A_1, \ldots, x_n\colon A_n \vdash \mathcal{I}$$

to the universally-quantified Pure implication

$$\bigwedge x_1, \ldots, x_n.\ [\![x_1\colon A_1; \ldots ; x_n\colon A_n]\!] \Longrightarrow \mathbf{enc}(\mathcal{I}),$$

where

$$\mathbf{enc}(t\colon T) := t\colon T, \quad \mathbf{enc}(a \equiv b\colon T) := a \equiv b$$

is the encoding of the typing and equality assertions previously discussed (Section 3.2). This translation is recursively extended to inference rules by defining

$$\mathbf{enc}\left(\frac{\mathcal{J}_1 \quad \cdots \quad \mathcal{J}_k}{\mathcal{J}}\right) := ([\![\mathbf{enc}(\mathcal{J}_1); \ldots ; \mathbf{enc}(\mathcal{J}_k)]\!] \Longrightarrow \mathbf{enc}(\mathcal{J}))\,.$$

Note that entailment and derivability are both translated to Pure implication, and that the order of variable typing assumptions is forgotten. Since we use the built-in Pure equality the only rules that need to be axiomatized for judgmental equality are $\Pi$- and $\Sigma$-congruence. The full list of logical rules is given in Figures 4 and 5. In the implementation, these rules are further organized into three named theorem collections `intros`, `elims` and `comps` for introduction, elimination and computation rules, facilitating their use by proof methods.

One needs to take care of the particular formulation of the rules one encodes, as a naïve translation into the Pure logic easily breaks adequacy (and by extension, soundness). This can be seen by the following example. Consider the standard formulation of the $\Pi$-introduction rule

$$\frac{\Gamma, x\colon A \vdash b\colon B}{\Gamma \vdash \lambda x\colon A.\, b\colon \prod x\colon A.\, B}$$

which would be encoded as

$$\big(\bigwedge x.\ x\colon A \Longrightarrow b(x)\colon B(x)\big) \Longrightarrow \lambda x\colon A.\, b(x)\colon \prod x\colon A.\, B(x)$$

according to the translation defined above. Taking this rule as an axiom would let us conclude the nonsense statement

$$\lambda x\colon \texttt{<}a,\, b\texttt{>}.\, \texttt{U(0)}\colon \prod x\colon \texttt{<}a,\, b\texttt{>}.\, \texttt{U(S(0))}$$

```
axiomatization where
```

PiF: $\llbracket \bigwedge x.\ x\colon A \Longrightarrow B(x)\colon \mathtt{U}(i);\ \ A\colon \mathtt{U}(i)\ \rrbracket \Longrightarrow \prod x\colon A.\, B(x)\colon \mathtt{U}(i)$

PiI: $\llbracket \bigwedge x.\ x\colon A \Longrightarrow b(x)\colon B(x);\ \ A\colon \mathtt{U}(i)\ \rrbracket \Longrightarrow \lambda x\colon A.\, b(x)\colon \prod x\colon A.\, B(x)$

PiE: $\llbracket\ f\colon \prod x\colon A.\, B(x);\ \ a\colon A\ \rrbracket \Longrightarrow f\ \grave{}\, a\colon B(a)$

beta: $\llbracket\ a\colon A;\ \bigwedge x.\ x\colon A \Longrightarrow b(x)\colon B(x)\ \rrbracket \Longrightarrow \lambda x\colon A.\, b(x) \equiv b(a)$

eta: $f\colon \prod x\colon A.\, B(x) \Longrightarrow \lambda x\colon A.\, (f\ \grave{}\, x) \equiv f$

Pi_cong: $\llbracket\ \ A\colon \mathtt{U}(i);$
$\quad \bigwedge x.\ x\colon A \Longrightarrow B(x)\colon \mathtt{U}(i);$
$\quad \bigwedge x.\ x\colon A \Longrightarrow B'(x)\colon \mathtt{U}(i);$
$\quad \bigwedge x.\ x\colon A \Longrightarrow B(x) \equiv B'(x)\ \rrbracket$
$\Longrightarrow \prod x\colon A.\, B(x) \equiv \prod x\colon A.\, B'(x)$

SigF: $\llbracket \bigwedge x.\ x\colon A \Longrightarrow B(x)\colon \mathtt{U}(i);\ \ A\colon \mathtt{U}(i)\ \rrbracket \Longrightarrow \sum x\colon A.\, B(x)\colon \mathtt{U}(i)$

SigI: $\llbracket \bigwedge x.\ x\colon A \Longrightarrow B(x)\colon \mathtt{U}(i);\ \ a\colon A;\ \ b\colon B(a)\ \rrbracket \Longrightarrow \texttt{<}a,\ b\texttt{>}\colon \sum x\colon A.\, B(x)$

SigE: $\llbracket\ \ p\colon \sum x\colon A.\, B(x);$
$\quad A\colon \mathtt{U}(i);$
$\quad \bigwedge x.\ x\colon A \Longrightarrow B(x)\colon \mathtt{U}(i);$
$\quad \bigwedge p.\ p\colon \sum x\colon A.\, B(x) \Longrightarrow C(p)\colon \mathtt{U}(i);$
$\quad \bigwedge x\, y.\ \llbracket x\colon A;\ \ y\colon B(x)\rrbracket \Longrightarrow f(x,y)\colon C(\texttt{<}x,\ y\texttt{>})\ \rrbracket$
$\Longrightarrow \mathtt{SigInd}(A,B,C,f,p)\colon C(p)$

Sig_comp: $\llbracket\ \ a\colon A;$
$\quad b\colon B(a);$
$\quad \bigwedge x.\ x\colon A \Longrightarrow B(x)\colon \mathtt{U}(i);$
$\quad \bigwedge p.\ p\colon \sum x\colon A.\, B(x) \Longrightarrow C(p)\colon \mathtt{U}(i);$
$\quad \bigwedge x\, y.\ \llbracket x\colon A;\ \ y\colon B(x)\rrbracket \Longrightarrow f(x,y)\colon C(\texttt{<}x,\ y\texttt{>})\ \rrbracket$
$\Longrightarrow \mathtt{SigInd}(A,B,C,f,\texttt{<}a,\ b\texttt{>}) \equiv f(a,b)$

Sig_cong: $\llbracket\ \ \bigwedge x.\ x\colon A \Longrightarrow B(x) \equiv B'(x);$
$\quad A\colon \mathtt{U}(i);$
$\quad \bigwedge x.\ x\colon A \Longrightarrow B(x)\colon \mathtt{U}(i);$
$\quad \bigwedge x.\ x\colon A \Longrightarrow B'(x)\colon \mathtt{U}(i)\ \rrbracket$
$\Longrightarrow \sum x\colon A.\, B(x) \equiv \sum x\colon A.\, B'(x)$

**Figure 4** Rules of Isabelle/Spartan: $\Pi$ and $\Sigma$ types.

**axiomatization where**

$$
\begin{array}{rl}
\mathsf{IdF}: & [\![\ A\colon \mathtt{U}(i);\ a\colon A;\ b\colon A\ ]\!] \Longrightarrow a =_A b\colon \mathtt{U}(i) \\
\mathsf{IdI}: & a\colon A \Longrightarrow \mathtt{refl}(a)\colon a =_A a \\
\mathsf{IdE}: & [\![\ \ p\colon a =_A b; \\
& \quad a\colon A; \\
& \quad b\colon A; \\
& \quad \bigwedge x\,y\,p.\ [\![p\colon x =_A y;\ x\colon A;\ y\colon A]\!] \Longrightarrow C(x,y,p)\colon \mathtt{U}(i); \\
& \quad \bigwedge x.\ x\colon A \Longrightarrow f(x)\colon C(x,x,\mathtt{refl}(x))\ ]\!] \\
& \Longrightarrow \mathtt{IdInd}(A,C,f,a,b,p)\colon C(a,b,p) \\
\mathsf{Id\_comp}: & [\![\ \ a\colon A; \\
& \quad \bigwedge x\,y\,p.\ [\![p\colon x =_A y;\ x\colon A;\ y\colon A]\!] \Longrightarrow C(x,y,p)\colon \mathtt{U}(i); \\
& \quad \bigwedge x.\ x\colon A \Longrightarrow f(x)\colon C(x,x,\mathtt{refl}(x))\ ]\!] \\
& \Longrightarrow \mathtt{IdInd}(A,C,f,a,a,\mathtt{refl}(a)) \equiv f(a)
\end{array}
$$

**Figure 5** Rules of Isabelle/Spartan: Identity type.

due to the semantics of the material implication, and the fact that the encoding does not *a priori* rule out ill-typed terms.[3] In order to prevent such occurrences, we add in typing *side conditions* to the premises of the inference rules, requiring the types and terms appearing in them to be well-typed. In the example above this entails adding the premise $A\colon \mathtt{U}(i)$ to obtain the rule PiI. This potentially creates more proof obligations for the user, but is mostly mitigated by automation.

## 4 The Isabelle/Spartan Implementation

### 4.1 Theorems and Proofs

Theorem goals in Spartan are stated using the Isar `schematic_goal` command instead of the usual `theorem` or `lemma` commands. This allows us to have schematic variables—Isabelle's notion of holes—in the goal state in order to support proof term synthesis, implicit arguments and term elaboration. In addition, the object logic provides schematic versions `theorem*`, `lemma*` and `corollary*` of the usual Isar goal commands.[4]

Proofs can be written in a mixture of tactic-style "apply"-proof scripts and declarative Isar, modulo a few technical issues. Most significantly, implicit arguments (Section 4.5), which are implemented using schematic variables, are forbidden from appearing in context assumptions declared using the `assume`/`assumes` Isar commands. Thus most of the time one would work using tactic-style proof scripts.

Isabelle provides a way to structure such scripts via the `subgoal` command. This, however, fixes any schematic variables in the goal, turning them into free variables, which interferes

[3] One underlying reason for this is our use of Russell-style universes, which forces the object terms and types to have the same syntactic category.
[4] Source code: `theorem_keywords.ML`.

**Table 1** Basic methods for Isabelle/Spartan.

| Method | Description |
|---|---|
| `typechk` | Solves rigid typing goals. |
| `rule` | Backwards reasoning: resolves the conclusion of a goal with the conclusion of a rule. Supports reasoning with types as propositions. |
| `intro`, `intros` | Refines the proof term by applying appropriate introduction rules (single and multi-step versions). |
| `reduce` | Computes terms; may fail if a term has not yet been sufficiently elaborated. |
| `equality` | Automates induction on identity types. |

with the synthesis of proof terms. Instead the object logic provides a new `focus` command[5] which focuses on the topmost subgoal in a proof while leaving schematic variables untouched, allowing us to synthesize specific subterms of the proof. The full syntax is

`focus` [`premises` [‹*hyps*›]] [`vars` ‹*var*›$^+$]

which also optionally moves goal premises out into the context and renames the fixed variables. The Coq-inspired bullet variations », ■, ▸ and ∼ are also provided to help further organize proofs.

Spartan extends the basic methods of Isabelle/Pure with the methods shown in Table 1.[6] These are set up to support the Isar "fixes–assumes–shows" style of goal statement, as opposed to writing the entire goal as a single Pure proposition. This parallels the distinction between the local context of hypotheses and the goal proper in dependently-typed systems.

## 4.2 Typechecking

The `typechk` method forms a key component of the automation provided by the logic, and is hooked into the other methods in Table 1 in order to automatically discharge the additional typing side conditions discussed in Section 3.5, which keeps them mostly hidden from the user. It is also installed as an additional simp-solver, which allows the Isabelle simplifier to handle typing conditions that arise in the course of simplifying terms. The method itself is implemented as a tactic that performs proof search using the type introduction and elimination rules, as well as any rules declared with the attribute `typechk`. However, it will only solve *rigid* typing goals, which are statements $t\colon T$ where the head of $t$ is not a schematic variable. This prevents it from wrongly instantiating schematic variables, which is especially important as it is also used to infer implicit arguments for term elaboration (Section 4.5).

## 4.3 Methods

The `rule` method performs backward resolution of the goal conclusion with given theorems. It extends the functionality provided by the Pure rule method in that it can also use rules whose propositional content is encoded as a type. That is to say, for example, that the result of the method invokations

```
apply (rule ‹⋀x y. x: A ⟹ y: B(x) ⟹ f(x,y): C(x,y)›)
```

[5] Source code: `focus.ML`.
[6] Source code: `tactics.ML`, `equality.ML`.

and

```
apply (rule ‹f: ∏x: A. ∏y: B(x). C(x,y)›)
```

on the goal

```
goal:
  ?prf: C(a,b)
```

is the same—the schematic variable $?prf$ is refined to $f(?a, ?b)$, subject to the new subgoals $?a\colon A$ and $?b\colon B(?a)$.

The `intro` and `intros` methods are used to refine proof terms; they are essentially `rule` restricted to resolving with the type introduction rules.

The `reduce` method performs simplification of terms by invoking the simplifier with type computation rules and other judgmental equalities declared `comp`. The method will occasionally fail if the term being reduced has not been sufficiently elaborated, which typically means that the typechecker is refusing to guess instantiations of implicit arguments.

## 4.4 Equality

Integrating the propositions-as-types approach to equality reasoning with the Isar language is slightly involved. This is best illustrated with an example. Suppose we wish to prove the transitivity of equality, which is naturally stated in Isar as shown in Listing 1.

**Listing 1** Transitivity of equality in Isar.

```
lemma* Id_transitive:
  assumes
    A: U(i)
    x: A
    y: A
    z: A
    p: x =_A y
    q: y =_A z
  shows
    ?prf: x =_A z
```

In a dependently-typed foundation, this statement corresponds to a judgment

$$x,\, y\, z\colon A,\ p\colon x = y \vdash ?prf\colon \textstyle\prod_{q\colon\, y=z} x = z$$

whose proof is straightforward: induct on $p$ to reduce the goal to inhabiting

$$\textstyle\prod_{q\colon\, x=z} x = z,$$

which is trivial. In Isar, however, the assumption $q\colon y = z$ is “free-floating” in the context, and we first have to “push it in” to the type of the conclusion to form the requisite dependent product before we can apply the identity elimination rule, after which point we want to “pull the new assumption” $q\colon x = z$ back out into the context. The `equality` method automates such translations between the dependently-typed and Isar formulations.

## 4.5 Term Elaboration

Spartan currently implements a simple semi-automated form of term elaboration, which works as follows. Holes for implicitly inferred arguments can be inserted into definitions using the syntax `?`, or directly into goal statements using `{}`. These are expanded into schematic variables by an automatic syntax phase translation[7] and hidden from the user with further syntax translations, only becoming visible when needed in proof goal obligations. Much of the time, however, many of these obligations are automatically solved by the typechecker tactic which is run after every call to a method provided by Spartan. Hence the user does not even see these obligations because they are automatically solved by the typechecker, which fills the hole with the inferred argument.

## 5 Practical Examples

In its current form, Isabelle/Spartan is already able to formalize nontrivial results from the Homotopy Type Theory book [13]. The following examples provide a demonstration of its capabilities and highlight opportunities for improvement.

### 5.1 Path Lifting

To demonstrate equality reasoning and term elaboration we prove the path lifting property.[8] This states that given a type family $P$ over $A$, $x$, $y$: $A$, $p$: $x = y$ and $u$: $P(x)$, we have that $(x, u) = (y, \mathrm{transport}^P(p, u))$.

```
lemma* pathlift:
  assumes
    A: U(i)
    ⋀x. x: A ⟹ P(x): U(i)
    x: A
    y: A
    p: x =_A y
    u: P(x)
  shows
    {}: <x, u> = <y, trans(P, p, u)>
```

The initial goal state is

```
goal:
  ?*1: <x, u> = <y, trans(P, p, u)>
```

Schematic variables beginning with `?*` arise from holes; in this case the one for the proof term. To begin, we apply induction on $p$:

```
apply (equality ‹p: _›)
```

The expression ‹$p$: _› is a direct *fact reference* that pattern matches to the context assumption $p\colon x =_A y$. Now the goal state reads

```
goal:
  ⋀x u. ⟦x: A; u: P(x)⟧ ⟹ ?b(x, u): <x, u> = <x, trans(P, refl(x), u)>
```

The typechecker has taken care of all the other side conditions, leaving us to focus on the core of the proof. The subexpression `trans`$(P,$ `refl`$(x), u)$ normalizes to $u$. Having proved this result earlier and declared it as a `comp` rule), we then

[7] Source code: `implicits.ML`.
[8] Source code: `Identity.thy`.

```
apply reduce
```

to further refine the goal, leaving

```
goal:
  ⋀x u. ⟦x: A;  u: P(x)⟧ ⟹ ?b(x,u): <x, u> = <x, u>
```

This is finished off with

```
apply intro
done
```

The Isabelle output panel now displays the synthesized proof term. By switching off implicit notation with the command

```
no_translations
  x = y  ↽  x =_A y
  trans(P,p) ↽ CONST transport(A,P,x,y,p)
```

we see that the $\Sigma$ type for the equality as well as the endpoints $x$ and $y$ of $p$ have been automatically inferred.

## 5.2 Bi- and Quasi-Inverses

Let us prove that bi-inverses are quasi-inverses. The proof will use almost all the features of Isabelle/Spartan that have been developed so far (with the exception of equality induction and universe level reasoning).[9]

The statement is

```
lemma* biinv_imp_qinv:
  assumes
    A: U(i)
    B: U(i)
    f: A → B
  shows
    {}: biinv(f) → qinv(f)
```

where `biinv` and `qinv` have been defined earlier in the usual way.

We begin with

```
apply intro
unfolding biinv_def
```

to pull the premise out into the context and unfold its definition. The goal now reads

```
goal:
  ⋀u. u: (∑g: B → A. g ∘ f ~ id_A) × (∑g: B → A. f ∘ g ~ id_B) ⟹ ?b(u): qinv(f)
```

Next we have to use the Pure `erule` method, together with an explicit call to typecheck side conditions,

```
apply (erule SigE, typechk)+
```

in order to split the sum in the premise into its components to obtain

[9] Source code: `Equivalence.thy`.

```
goal:
  ⋀u u′ y g y′ g′ y″. ⟦g: B → A;  y′: g ∘ f ∼ id_A;  g′: B → A;  y″: f ∘ g′ ∼ id_B⟧
     ⟹ ?f(u, u′, y, g, y′, g′, y″): qinv(f)
```

The universally-quantified variables $u'$, $y$ that have appeared come from the unused names in the nondependent function type of $g, g' \colon B \to A$, and can be safely ignored. The basic Pure method `erule` performs *elim-resolution* which matches on the term $u$ we want to eliminate from the premise, but does not automatically handle side conditions.

In order to continue we will need to explicitly refer to the newly-separated components. We use the `focus` command to name them and pull their properties out of the goal statement and into the Isar context:

```
focus premises vars _ _ _ g H1 h H2
```

obtaining

```
goal:
  ?f(u_, u′_, y_, g, H1, h, H2): qinv(f)
```

The universally-quantified variables have now been fixed, and Isabelle automatically invents names for the variables we left unspecified. The properties $\texttt{g}, \texttt{h} \colon B \to A$, $\texttt{H1} \colon \texttt{g} \circ f \sim \mathrm{id}_A$ and $\texttt{H2} \colon f \circ \texttt{h} \sim \mathrm{id}_B$ can now be explicitly referenced.

Now we can prove that $f$ is a quasi-inverse. First we unfold the definition of `qinv` and refine it with

```
unfolding qinv_def
apply intro
```

to obtain the two proof obligations

```
goal (2 subgoals):
  1. ?a(u_, u′_, y_, g, H1, h, H2): B → A
  2. ?b(u_, u′_, y_, g, H1, h, H2):
        (?a(u_, u′_, y_, g, H1, h, H2) ∘ f ∼ id_A) × (f ∘ ?a(u_, u′_, y_, g, H1, h, H2) ∼ id_B)
```

It helps the proof structure here to use focus bullets. Thus

```
■ by (rule ‹g: _›)
```

finishes the first subgoal and instantiates the schematic variable $?a$ with `g`, leaving

```
goal:
  ?b(u_, u′_, y_, g, H1, h, H2): (g ∘ f ∼ id_A) × (f ∘ g ∼ id_B)
```

We refine the conjunction and prove the first conjunct with

```
■ apply intro
  apply (rule ‹H1: _›)
```

Finally, it only remains to prove the second conjunct. This is done by constructing a sequence of homotopies ultimately showing that $\texttt{g} \sim \texttt{h}$, from which, together with $\texttt{H2} \colon f \circ \texttt{h} \sim \mathrm{id}_B$, the result follows. This is best done in a "calculation" proof block, the outline of which is

```
proof -
  have ?α: g ∼ g ∘ f ∘ h
```

```
        ‹proof›
      moreover have ?β: g ∘ f ∘ h ∼ h
        ‹proof›
      ultimately have ?γ: g ∼ h
        ‹proof›
      then have ?δ: f ∘ g ∼ f ∘ h
        ‹proof›
      thus {}: f ∘ g ∼ id_B
        ‹proof›
    qed
```

The details of the subproofs are not very enlightening and are hence omitted. They mostly consist of rewriting with the basic Pure `subst` method and properties of homotopy, together with typechecking. Having proved this final subgoal we obtain the fully-elaborated proof term, which is displayed in the output panel.

## 6 Conclusion and Future Work

The implementation of Isabelle/Spartan shows that, despite being built on simple type theoretic foundations, Isabelle's logical framework infrastructure is feasibly able to support the development and use of dependent type theory as a proof environment.

Many improvements to Spartan are possible: there is currently no general elimination tactic, definitions for synthesized proof terms should be exported for reusability, the rudimentary typechecking mechanism can likely be improved, and automation to enable universe ambiguity might prove desirable. Ongoing and future work aims to implement these improvements, as well as to establish theoretical properties of the encoding, expand the collection of basic types and the library of formalizations, and explore how the framework, along with techniques discussed in this paper, may be used to implement related logics like the calculus of inductive constructions, two-level type theory [2] and cubical type theory [6].

## Acknowledgements

I thank Makarius Wenzel for multiple discussions explaining technical details of the Isabelle system, and Cezary Kaliszyk for numerous helpful suggestions for improving this paper.

## References

1 Dedukti - a Logical Framework, 2020. Accessed 2020-02-13. URL: `https://deducteam.github.io/`.

2 Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-Level Type Theory and Applications, 2017. `arXiv:1705.03307`.

3 Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Dedukti: a Logical Framework based on the λΠ-Calculus Modulo Theory. 2020. URL: `http://www.lsv.fr/~dowek/Publi/expressing.pdf`.

4 Andrej Bauer. Spartan type theory. Talk given at the School and Workshop on Univalent Mathematics, University of Birmingham, December 2017. URL: `http://math.andrej.com/2017/12/11/spartan-type-theory/`.

5 Andrej Bauer, Gaëtan Gilbert, Philipp G. Haselwarter, Anja Petković, Matija Pretnar, and Christopher A. Stone. The Andromeda proof assistant. `https://www.andromeda-prover.org/`, 2020. Accessed 2020-02-13.

6 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: a constructive interpretation of the univalence axiom, 2016. `arXiv:1611.02108`.

7 Bart Jacobs and Tom Melham. Translating Dependent Type Theory into Higher Order Logic. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, pages 209–229. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993. `doi:https://doi.org/10.1007/BFb0037108`.

8 Lawrence C. Paulson. Isabelle/CTT. Isabelle object logic, bundled with the Isabelle proof assistant distribution, 1991. URL: `https://isabelle.in.tum.de/website-Isabelle2019/dist/library/CTT/index.html`.

9 Lawrence C. Paulson. *Isabelle's Logics*, June 2019. URL: `https://isabelle.in.tum.de/website-Isabelle2019/dist/Isabelle2019/doc/logics.pdf`.

10 Frank Pfenning. Elf: A meta-language for deductive systems. In Alan Bundy, editor, *Automated Deduction — CADE-12*, pages 811–815. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994. `doi:https://doi.org/10.1007/3-540-58156-1_66`.

11 Frank Pfenning. Logical Frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, Handbook of Automated Reasoning, chapter 17, pages 1063–1147. North-Holland, Amsterdam, 2001. `doi:https://doi.org/10.1016/B978-044450813-3/50019-9`.

12 Frank Pfenning and Carsten Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *Automated Deduction — CADE-16*, pages 202–206. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. `doi:https://doi.org/10.1007/3-540-48660-7_14`.

13 Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* 2013. URL: `https://homotopytypetheory.org/book`.

14 University of Cambridge, Technische Universität München, and contributors. Isabelle2019, June 2019. URL: `https://isabelle.in.tum.de/website-Isabelle2019/`.

15 Makarius Wenzel. *The Isabelle/Isar Reference Manual*, June 2019. URL: `https://isabelle.in.tum.de/website-Isabelle2019/dist/Isabelle2019/doc/isar-ref.pdf`.

16 Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle Framework. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 33–38. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

17 Markus Wenzel. Isar - A Generic Interpretative Approach to Readable Formal Proof Documents. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, volume 1690 of *Lecture Notes in Computer Science*, pages 167–184. Springer, 1999. `doi:10.1007/3-540-48256-3\_12`.