Towards a Mechanization of Standard ML in Beluga using Harpoon

Marc-Antoine Ouimet Jacob Errington Brigitte Pientka
McGill University, Montreal, Canada
{marc-antoine.ouimet,jacob.errington}@mail.mcgill.ca bpientka@cs.mcgill.ca

The metatheory of Standard ML was formalized in LF and verified in Twelf by Lee, Crary and Harper. We set out to repeat their meta-theoretic mechanization in order to assess the practicality and robustness of Beluga, a programming and proof environment with support for first-class contexts and simultaneous substitutions, using Harpoon, an interactive proof environment acting as a more accessible frontend to Beluga. We report on the progress made in translating the mechanization, and present issues and missing features from Beluga and Harpoon that need to be addressed before resuming work on the translation.

1 Introduction

The mechanization for the metatheory of Standard ML presented by Lee, Crary and Harper in 2009 was the first of its kind as it formalized a full-fledged general purpose programming language [9]. Indeed, Standard ML is a dialect of ML featuring higher-order functions, algebraic datatypes, pattern matching, polymorphism, as well as a rich module system. The mechanization spanned many years of research into providing a type-theoretic definition of Standard ML [7], formalising its module system [6], and devising verification techniques when dealing with dependent types and subkinding relations [2, 18, 20]. Other programming languages, such as Java [8], C [11] and OCaml [12], have had subsets of their features formalized, though not all without mishaps, which goes to show how difficult of a task it is correctly and convincingly mechanize a programming language.

The mechanization of Standard ML was realised in Twelf [13], a meta-logical framework for deductive systems based around the LF type theory [5]. Among the limitations of Twelf are its lack of support for substitutions and reasoning explicitly with contextual types, which makes proof development less straightforward. Beluga [16] took inspiration on Twelf and augmented it with support for first-class contexts and simultaneous substitutions, among other things. Since Twelf and Beluga share the same LF core, it followed that the mechanization of Standard ML could be translated into Beluga to see how it could be improved upon with a language supporting first-class contexts. This was also the opportunity to employ Harpoon [4], an interactive command-driven proof environment

built on top of Beluga, to evaluate the robustness and practicality of both Beluga and Harpoon.

In this paper, we report on missing features and issues with Beluga and Harpoon that were encountered in the process of translating the mechanization of Standard ML. These range from issues arising from Beluga's lack of support for existentially-quantified metavariables to stringent context subsumption errors that prevent certain programs to be checked, as well as missing support for complete inductive reasoning. Additionally, inconsistencies in translating from Harpoon proof scripts to Beluga programs revealed issues with the totality checking procedure that allow for incorrect proofs to be included in the translated mechanization.

2 Overview of the mechanization

This section presents the key components and structure of the original mechanization and reports on the progress made in translating it to Beluga.

2.1 Structure of the mechanization

Lee, Crary and Harper's mechanization of Standard ML [9] encompasses many research efforts with the formalization of dependent types and modularity, and the implementation of the TILT compiler. As such, the many languages formalized therein have evolved greatly from when they were first presented. It employs a core calculus approach, whereby well-behaved lambda calculi with sufficient expressive power are elaborated to the abstract syntax of the actual language. This mechanization may be split into three components:

1. The singleton type calculus [3, 18, 20], referred to as singleton metatheory.
2. An explicitly-typed λ-calculus, referred to as internal language [9], which includes type constructors, dependent types, signatures, stores, terms and modules.
3. An elaboration and proof of correctness from the internal language to the abstract syntax of Standard ML [10], referred to as external language [9].

Type preservation for terms of the internal language requires proofs that some type constructors for simple types are injective. These include product, sum, arrow, recursive and labelled types. In the presence of dependent kinds and a type subsumption rule, these injectivity lemmas are not trivial to prove. As such, the singleton metatheory, with its extensional equivalence relations, was proven to be consistent with those types of the internal language in order to prove their injectivity. Stone and Harper [20] proposed an algorithm

involving logical relations for proving soundness and correctness of the singleton kind calculus, but the original mechanization instead features a syntactic proof [3] due to limitations of Twelf. Using Beluga's inductive and stratified types as in the case studies on logical relations [1], it should be possible to mechanize the algorithm they had found initially, which may shorten the mechanization.

The internal language is the core calculus of the mechanization [9]. Progress for terms and modules are proven by reducing them first to their canonical forms. Similarly, inversions are proven by reducing the kinds of constructors for terms and modules to their most specific kind [19]. The syntax and semantics of the internal language appear in appendix A, and figure 1 summarizes its properties and their dependencies in proving its type safety.

The internal language is then finally elaborated to assert its regularity against the abstract syntax of Standard ML. This part of the mechanization follows closely the formal definition of Standard ML [9] and provides a benchmark for LF parsers and type reconstructors. An adaptation of this elaboration in Beluga would follow closely that of the original mechanization.

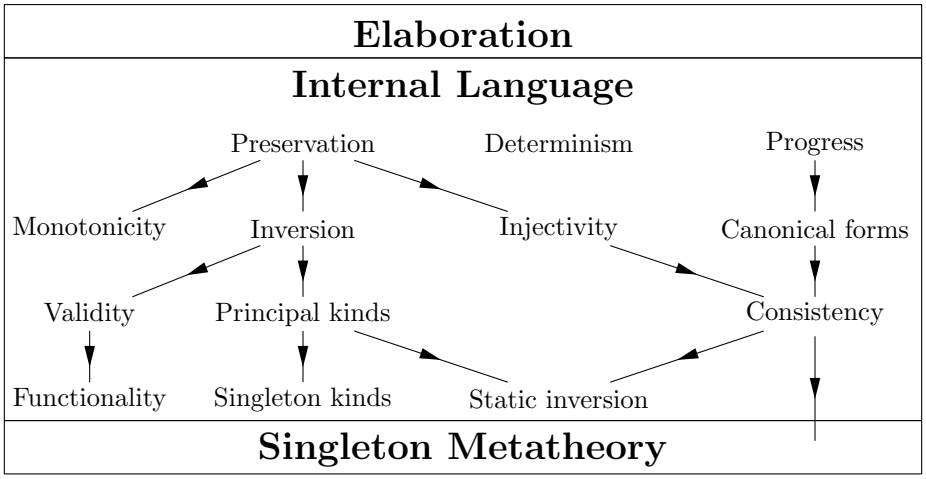


Figure 1: Overview dependency graph of properties and theorems involved in proving the type safety of the internal language. Type preservation requires the property that stores evolve monotonically, that constructors for simple types are injective, and that constructors for terms and modules are invertible. erm and module constructors are invertible. These inversion lemmas themselves require notions of equivalence for well-formed kinds and type constructors as well as module signatures. Functionality lemmas assert that the equivalence for these type families are maintained when substituting equivalent objects. Once these properties are established, the internal language may be elaborated into the abstract syntax of Standard ML. Notable lemmas from the internal language are stated in appendix B.

2.2 Translation progress

We set out to translate the proof of type safety of the internal language using a top-down approach. This involved among other things translating the validity, or well-formedness, of equivalence relations for kinds, type constructors and signatures, as well as the validity of subkinding. Functionality lemmas on which these depend have not been translated since the issue with dependently-kinded constructors and explicit contexts arose [9], whereby an invariant on the position of a contextual assumption cannot be maintained using hypothetical judgments when dealing with dependent types. We have yet to see how to adapt the heavy encoding technique of explicit contexts [2] in Beluga's notion of inductive types to yield smaller programs.

Injectivity lemmas for type constructors are required for type preservation of terms in the internal language. Their proofs as proposed in the original mechanization [9] make use of the singleton metatheory for its unary and binary equivalence algorithms [20]. These have yet to be translated since they depend on completeness theorems of other calculi whose translations from one to the other have not been added. However, focus has been given to translating the consistency theorems between the singleton metatheory and the kinds and type constructors of the internal language.

The translated mechanization need not feature simple substitution nor equality lemmas as Beluga features simple and simultaneous substitutions. LF constants for variable terms, modules and type constructors were removed as these are handled using contextual types. Empty LF types exclusively used for contextual assumptions were also removed. Specifically, in the internal language, the assumption judgments that a term has a given type constructor, and that a module has a given signature had their usages replaced with higher-order typing judgements. Indeed, these assumptive judgements were not constrained to specific store types in order to satisfy their weakening property with respect to heap and tag types [9]. Since typing judgments for terms and modules include a store type explicitly, then we universally quantify over the store type using higher-order contextual types as illustrated in figure 2. A substitution lemma as found in the original mechanization is then required to satisfy these hypothetical judgments with a type judgment bound to a specific store type.

3 Verification with Beluga

This section presents issues with Beluga that need to be addressed in order to enhance, fix and verify the translated mechanization.

```
% tm-assm : term → con → type.
% tm-of : sttp → term → con → type.
tm-of/lam :
  cn-of T1 t →
  ({x : term} tm-assm x T1 → tm-of F (E x) T2) →
    tm-of F (tm/lam T1 ([x] E x)) (arrow T1 T2)
tm-of/lam' :
  cn-of T1 t →
  ({x : term} {u : {F' : sttp} tm-of F' x T1} tm-of F (E x) T2) →
    tm-of F (tm/lam T1 (\x. E x)) (arrow T1 T2)
```

Figure 2: Typing judgment for abstraction of terms in the internal language. The constructor `tm-of/lam` features an empty LF constant `tm-assm` which is not constrained to a store type, and the constructor `tm-of/lam'` adapts this hypothesis judgment with a Π-type and reusing the `tm-of` type.

3.1 Mode information

This subsection highlights how Beluga would benefit from having a feature analogous to Twelf's mode declaration for specifying existentially-quantified metavariables.

Beluga currently does not support Σ-types as they pose implementation problems. Instead, the documentation proposes that theorems involving existential quantification be skolemised using Π-types and an auxiliary LF type with a unique constructor [17]. Skolemisation reduces the readability of theorem statements and subsequent subgoals in Harpoon sessions, and would pose problems in efforts to mechanize the translation of Beluga programs into human-readable proofs as it erases the existential quantification intent. An example of such a workaround is given in figure 3.

$$\frac{\Gamma \vdash \texttt{C : con}}{\Gamma \vdash \Sigma \texttt{M:eterm.map C M : type}}\ \text{can-map}$$

```
LF can-map/e : con → type = can-map/i : {M : eterm} map C M → can-map/e C;
proof can-map : (g : can-map-ctx) {C : [g ⊢ con]} [g ⊢ can-map/e C] = % ...
```

Figure 3: Derivation involving existential quantification, and equivalent theorem statement in Beluga using an auxiliary LF type and constructor. After invoking `can-map`, a metavariable `M : eterm` and the judgement `map C M` are obtained by inversion on the `can-map/e` judgment.

Twelf supports the statement of ∀∃-metatheorems with multiple outputs through modes [14]. This allows for multiple judgments to be proved simultaneously, which is convenient when their proof structures are similar, or depend on each other. Subsequent uses of such theorems in other proofs may introduce metavariables that go unused. However, grouping of similar theorems with this mode declaration prevents the clutter of having theorem names for each output.

Nevertheless, there are theorems that require multiple outputs. These involve existentially-quantified metavariables that appear in more than one output judgment. Attempting to separate these outputs results in weaker and non-equivalent statements. Indeed, since these are invoked separately, the output metavariables that were existentially-quantified are not necessarily equal. Hence, such theorems' outputs need to be tuples that are destructured upon invocation. An example of a theorem requiring multiple outputs is given in figure 4.

$$\frac{\Gamma \vdash \texttt{tm-of F (tm/pair E1 E2) T : type}}{\Gamma \vdash \Sigma \texttt{T1:con}.\Sigma \texttt{T2:con}.\begin{pmatrix}\texttt{cn-equiv (prod T1 T2) T t : type}\\ \texttt{tm-of F E1 T1 : type}\\ \texttt{tm-of F E2 T2 : type}\end{pmatrix}}\ \text{inversion-tm/pair}$$

```
LF inversion-tm/pair/e : sttp → term → term → con → type =
  inversion-tm/pair/i : {T1 : con} {T2 : con} cn-equiv (prod T1 T2) T t →
    tm-of F E1 T1 → tm-of F E2 T2 → inversion-tm/pair/e F E1 E2 T;
proof inversion-tm/pair : [ ⊢ tm-of F (tm/pair E1 E2) T] →
    [ ⊢ inversion-tm/pair/e F E1 E2 T] = % ...
```

Figure 4: Derivation involving existential quantification and multiple outputs, and equivalent theorem statement in Beluga using skolemisation. With this formulation, destructuring of what would be the output tuple is obtained by inversion on the `inversion-tm/pair/e` judgment.

3.2 Context schemas overhaul

Assumptions in Twelf require the declaration of regular worlds to ensure proper type inference when dealing with hypothetical judgments arising from LF constructors [14]. Beluga adopts a similar solution where it uses its notion of context schemas to enable the specification of its first-class contexts. However, this feature currently lacks some of the functionalities of regular world declarations.

Beluga does not allow the definition of schemas with alternating assumptions by referring to pre-existing schemas, which affects the maintainability of mechanizations involv-

ing multiple arrangements of schemas. In other words, schema definitions cannot refer to previous schemas to append or prepend them to others to form alternating assumptions. Implementing this feature would allow for context schemas to not feature duplicate schema specifications. Figure 5 illustrates how this feature may be implemented by overriding the plus operator for defining alternating assumptions.

```
schema conblock = block (a : con);
schema termblock = block (x : term);
schema modblock = block (a : con, m : module', dfst : md-fst m a);
schema resolve-ctx = conblock + termblock + modblock;
schema resolve-ctx' = conblock + block (x : term) + modblock;
```

Figure 5: Proposed change to the alternating schema operator. Schemas `resolve-ctx` and `resolve-ctx'` are equivalent.

Beluga does not strictly ensure that all indeterminate variables in schema declarations are present in the meta-context when instantiating open assumption blocks from context schemas that have such variables. That is, the metavariables from the `some` clause of a context schema declaration are not all ensured to exist when instantiating the assumption. In the mechanization, this has led to some theorems invoking lemmas requiring the well-formedness of kinds to disregard the well-formedness judgment entirely. Specifically, referring to the context schemas illustrated in figure 6, this has led to the erroneous invocation of lemmas in the context `conbind-reg` while only meeting the requirements from `conbind`.

```
schema conbind = some [K : kind] block (a : con, d : cn-of a K);
schema conbind-reg = some [K : kind, wf : kd-wf K]
  block (a : con, da : cn-of a K);
```

Figure 6: Schema declarations that bind type constructors to kinds. Schema `conbind-reg` should be usable in place of `conbind` since the former is weaker as the latter does not require the kind to be well-formed. The converse does not hold.

Beluga does not treat context schemas as lattices. Context subsumption ensures that if a schema Γ is a prefix of a schema Γ', then Γ' can be used in place of Γ. Although this is already featured in Beluga with context schemas consisting of only closed assumption blocks, it is not the case for open assumption blocks. Figure 7 illustrates an unsupported context subsumption involving indeterminate variables that are expected to agree with each other. Therein, the judgments required by the `some` clause to make an assumption in `unmap-bind` are stricter but include those required for assumptions made in `conbind-reg`. To circumvent

this in the translation, stronger context schemas were prepended to those with which they did not agree. This has led to more general theorem statements which may not be provable as bound variables implicit to the stronger schemas may not feature required judgments.

```
schema conbind-reg = some [K : kind, wf : kd-wf K]
  block (a : con, da : cn-of a K);
schema unmap-bind = some [K : kind, wf : kd-wf K, B : etp, map : tunmap B K]
  block (a : con, da : cn-of a K, x : eterm, dx : eof x B, xt : unmap x a);
```

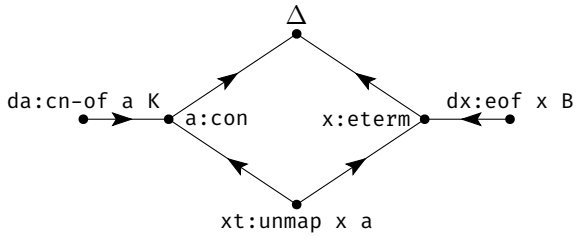
Figure 7: Context schema binding well-formed kinds to type constructors, and context schema mapping types from the singleton metatheory to kinds in the internal language. `unmap-bind` should be usable in place of `conbind-reg` since the latter is weaker and its assumption requirements are included in the former.

Beluga attempts to generate all possible valid appeals to the induction hypothesis in its termination checking algorithm for recursive programs. However, it does not take into account all admissible orderings of judgments in block assumptions and contexts. Any topological ordering of the dependency graph of judgments in these objects should be admissible. For instance, figure 8 illustrates different assumption blocks of topologically-ordered judgments which are expected to agree with a declared context schema. These topological orderings should also apply when appealing to an induction hypothesis, as illustrated in figures 9 and 10.

3.3 Complete induction

Totality checking in Beluga does not support specifying that the output of a lemma is non-inflationary, which prevents it from being used recursively in other programs. Twelf supports this using its reduction declaration feature [14]. In the original mechanization, these reduction declarations are used among other things to verify that the outputs of respective equality lemmas are of the same order as that of their inputs. However, these lemmas may be inlined in Beluga by inversion on each of the LF constructors for equality types. More interestingly, the proposed proofs for certain type constructor inversion theorems make use of the reduction declaration in one of their invoked lemmas to work around the subsumption relation on type constructors. The reduction declaration then obviates the need to inline the invoked lemma, whose proof is more intricate and whose duplication would affect the maintainability and readability of the mechanization. This feature may be worth implementing in Beluga as it plays a role in proving canonical form lemmas for modules later in the mechanization. For this, the current totality checking algorithm may need to be revisited to

```
schema unmap-bind =
  some [K : kind, wf : kd-wf K,
        B : etp, map : tunmap B K]
  block (a : con, da : cn-of a K,
         x : eterm, dx : eof x B,
         xt : unmap x a);
```




```
block (a : con, da : cn-of a K, x : eterm, dx : eof x B, xt : unmap x a);
block (x : eterm, dx : eof x B, a : con, da : cn-of a K, xt : unmap x a);
block (a : con, x : eterm, xt : unmap x a, da : cn-of a K, dx : eof x B);
```

Figure 8: Context schema declaration, dependency graph of the assumption judgments, and list of equivalent assumption blocks with different topological orderings. The node Δ in the dependency graph stands for the indeterminate variables from the some clause on which the assumption judgments depend, namely `K : kind` and `B : etp`. Each of the listed blocks is expected to agree with `unmap-bind`. Since the judgments in Δ are expected to be in the meta-context upon instantiation of the assumption block, they should already be present in some topological ordering, and so the orderings of the judgments in the some and block clauses are independent.

```
schema termbind = some [T : con] block (x : term, u : {F' : sttp} tm-of F' x T);
proof substitution-tm-tm : (g : termbind)
  [g, u : {F' : sttp} tm-of F' E1[..] T1[..] ⊢ tm-of F[..] E2[..] T2[..]] →
  [g ⊢ tm-of F E1 T1] → [g ⊢ tm-of F E2 T2] = % ...
% Meta-context: {intros ← split x1 (case tm-of/try)}
D2 : (g, u : {F' : sttp} tm-of F' (E[..]) (T[..]),
        x : term, dx : {F' : sttp} tm-of F' x tagged ⊢
          tm-of (F[..]) (E1[.., x]) (T1[..]))
D : (g ⊢ tm-of F E T)
% Harpoon command prompt:
by substitution-tm-tm [_, b : block (x : term, u : {F' : sttp} tm-of F' x _),
  u : {F' : sttp} tm-of F' _ _ ⊢ D2[.., u, b.x, b.dx]]
  [_, b ⊢ D[..]] as D2' unboxed
```

Figure 9: Unsupported exchange of contextual objects. The input Harpoon command raises an error since it doesn't find an equivalent induction hypothesis among those Beluga generated.

```
schema can-map-ctx = block (x : eterm)
  + some [x : eterm] block (a : con, at : map a x);
LF can-map/e : con → type = can-map/i : {M : eterm} map C M → can-map/e C;
proof can-map : (g : can-map-ctx) {C : [g ⊢ con]} [g ⊢ can-map/e C] = % ...
% Meta-context: {intros ← split [g ⊢ C] (case rec')}
C1 : (g, a : con, b : con ⊢ con)
% Harpoon command prompt:
invert can-map
  [g, x : eterm, b1 : block (a : con, d : map a x),
      y : eterm, b2 : block (b : con, e : map b y) ⊢ C1[.., b1.a, b2.b]]
```

Figure 10: Unsupported appeal to the induction hypothesis when dealing with a context of alternating assumptions.

incorporate ideas from Twelf's implementation for such verification of complete induction [15].

4 Correctness of Harpoon sessions

This section presents issues with Harpoon that need to be addressed in order to verify the translated mechanization.

4.1 Non-exhaustive case analyses

Manually editing Harpoon proof scripts reveals that they are not checked for non-exhaustive case analyses. Although Harpoon's tactics do produce exhaustive analyses, certain theorems in the translation encountered internal errors when reconstructing cases for bound variables implicit to a context variable with a schema of alternating assumptions, or one containing higher-order assumptions. In order to continue with the mechanization, these cases were left out to be completed at a later time. Should these cases remain omitted, the proof should be invalid and fail type reconstruction upon completion of the other cases. Thus, such proofs should explicitly state that they do not meet the totality checker's requirements using the `trust` keyword in place of the induction order.

4.2 Inconsistencies with Beluga programs

There are technical discrepancies between Harpoon's and Beluga's syntaxes for programs which prevent Harpoon proof scripts to be correctly translated into Beluga programs. Notably, Harpoon's type erasure procedure is too aggressive and removes types from contextual objects in such a way that they may not have their types reconstructed by Beluga. Further, Harpoon proof scripts involving context schemas with alternating assumptions and higher-order contextual objects fail to be reconstructed. The error with respect to alternating schemas may be due to having duplicate names for case analyses on variables bound in the context. As for higher-order contextual objects, these raise ill-typed expression in the computational context upon type reconstruction. Resolving these issues is critical for our attempt at replacing the empty LF constructors introduced as assumption judgments from the original mechanization with variables bound in the context of the proof.

Scoping of metavariables in Harpoon is inconsistent in-between sessions. Specifically, variables which were not in scope unexpectedly become in scope after reconstructing a serialized session. As such, sessions resumed using a different instance of Harpoon may incorrectly refer explicitly to metavariables that are not in scope, in which case the translated Beluga programs fail to be verified. When out of scope metavariables appear as contextual objects and variables, they usually need to be reconstructed by Beluga, hence they are left out as holes instead of being incorrectly referred to explicitly. This is further problematic as these holes can be silently misused.

4.3 Uninstantiated holes

Proofs completed in Harpoon and checked with Beluga may wrongly contain uninstantiated metavariables. These arise when holes for variables or contextual objects that would not appear in the translated Beluga program are not present in a subgoal's meta-context when completing a proof. Such holes should only be used where type-checking is possible, and since these lie outside of the Harpoon's session context by the end of the proof, they avoid being checked. This may lead to users unknowingly misuse holes as this type of error is not caught upon type reconstruction in subsequent Harpoon sessions. Figures 11 and 12 illustrate incorrect proofs that are not caught by the checking algorithm.

Resolving these sorts of issues with uninstantiated holes in proof scripts during a Harpoon session would require commands to navigate through and edit proof trees. If navigation through proof trees were to be implemented, then it would also be interesting to add the ability to check proofs downwards from a given step. Since large mechanizations contain proofs with multiple branches, partial verification from a given node in a proof would allow users to detect errors before completing the entire proof.

Theorems may not have their proofs completed in Harpoon without having uninstan-

```
1  rec preservation-tm : [ ⊢ tm-of F E T] → [ ⊢ store-of F ST F] →
2    [ ⊢ step ST E ST' E'] → [ ⊢ preservation-tm/e F E' T ST'] =
3  fn y ⇒ fn x ⇒ fn z ⇒
4    let [ ⊢ Dof] = y in
5    let [ ⊢ Dstof] = x in
6    case z of
7    % ...
8    | [ ⊢ step/lett2 Dvalue] ⇒
9      let [ ⊢ inversion-tm/lett/i _ Dof1 (\x. \u. Dof2)] =
10         inversion-tm/lett [ ⊢ Dof] in
11     let [ ⊢ Dof2'] = [ ⊢ Dof2[_, (\f. _)] ] in
12     [ ⊢ preservation-tm/i _ Dof2' Dstof (extends/nil )];
```

Figure 11: Invalid Beluga program translated by Harpoon in an attempt to prove type preservation when stepping into a (**let** `E1` **in** `E2`)-expression. The highlighted segment of line 11 incorrectly eliminates the Π-type for the assumption on the type of `E1` being invariant of the store type in order to substitute it in `E2`. This elimination of the Π-type requires a separate lemma.

tiated metavariables because schemas in Beluga are not considered as lattices. That is, without the notion of topological orderings of assumption blocks instantiated from context schemas, the current implementation of context subsumption does not allow for topologically-ordered subsets of assumptions to agree with other context schemas. This also implies that assumptions unrelated to the output of a theorem may not be eliminated by the theorem prover. In Harpoon, this has led to inconsistencies in the contextual objects of a theorem's output when it is stated using an auxiliary LF type with a unique constructor as opposed to being stated in a straightforward fashion. The former statement of such a theorem is much like that of those in Skolem normal form as featured in section 3.1, although there may not be a need for explicit Π-types in the auxiliary LF constructor. Figure 13 illustrates how different results are obtained when stating the same theorem in the two forms. The distinction worth noting between the two is that the straightforward statement of the theorem does not explicitly flag the last judgment as an output. Performing inversion on the auxiliary LF constructor emulates Twelf's mode information for specifying judgments as outputs.

4.4 Variable cases reconstruction

Harpoon's proof serialisation algorithm mishandles variable cases in the presence of alternating assumptions schemas. When performing case analysis on a type in the presence of

```
1  rec functionality-kd-reg : (g : conbind)
2    [g, b : con, db : cn-of b K[..] ⊢ kd-wf K'[.., b]] →
3    [g ⊢ cn-equiv C1 C2 K] → [g ⊢ cn-of C1 K] →
4      [g ⊢ kd-equiv K'[.., C1] K'[.., C2]] =
5  fn z ⇒ fn y ⇒ fn x ⇒
6    case z of
7    % ...
8    | [g, b, db ⊢ kd-wf/sigma Dwf1 (\a. \da. Dwf2)] ⇒
9      let [g ⊢ Dequiv1] =
10         functionality-kd-reg [g, b : con, db : cn-of b _ ⊢ Dwf1] y x in
11     let [g ⊢ Dequiv] = y in
12     let [g ⊢ Dof] = x in
13     let [g, x ⊢ Dequiv2] =
14         functionality-kd-reg
15           [g, x : block (a : con, da : cn-of a _),
16               b : con, db : cn-of b _ ⊢ Dwf2[.., b, db, x.1, _]]
17           [g, x : block (a : con, da : cn-of a _) ⊢ Dequiv[..]]
18           [g, x : block (a : con, da : cn-of a _) ⊢ Dof[..]] in
19     [g ⊢ kd-equiv/sigma Dequiv1 (\a. \da. Dequiv2[.., <a; da>])];
```

Figure 12: Invalid Beluga program translated by Harpoon in an attempt to prove the functionality lemma for regular kinds, without specifying the invariant on the position of contextual assumptions. The highlighted segment of line 16 incorrectly performs an exchange on the order of type constructors in the context. In the presence of dependent kinds, this swap is not sound as the context is no longer in a topological ordering because `a`'s kind may depend on `b`. Hence, type reconstruction should fail to instantiate the hole from line 16. A proof involving explicit contexts [2] should be provided instead.

context schema declaration, it is possible for that type to originate from the context. This leads to the generation in Beluga of an additional case referred to as the variable case. Since contextual assumptions are grouped together in sum types, then variable cases specifically refer to type projections. As such, the current implementation of Harpoon uses the index of the analysed type in the assumption block as naming scheme for variable cases. This leads to naming conflicts whereby multiple branches of a proof share the same case identifier when dealing with alternating assumptions schemas, in which case type reconstruction fails when the meta-contexts are parsed and serialized in different orders. Errors raised by Harpoon in the presence of variable cases include ill-type expression errors, internal errors and type

```
schema map-bind = some [B : etp, K : kind, Dmap : tmap K B, Dwf : ewf B]
  block (x : eterm, dx : eof x B, a : con, da : cn-of a K, at : map a x);
proof map-wf : (g : map-bind) [g ⊢ kd-wf K] →
  [g ⊢ tmap K A] → [g ⊢ ewf A] = % ...

LF map-wf'/e : etp → type =  map-wf'/i : ewf A → map-wf'/e A;
proof map-wf' : (g : map-bind) [g ⊢ kd-wf K] →
  [g ⊢ tmap K A] → [g ⊢ map-wf'/e A] = % ...

% Dwf : (g, a : con, da : cn-of a K[..] ⊢ kd-wf K'[.., a])
% Dtmap : (g, a : con, x : eterm, at : map a x ⊢ tmap K'[.., a] A[.., x])

by map-wf [g, b : block (
    x : eterm, dx : eof x B[..], a : con, da : cn-of a K[..], at : map a x
  ) ⊢ Dwf[.., b.a, b.da]] [_, b ⊢ Dtmap[.., b.a, b.x, b.at]] as Dwf' unboxed
%{ ⇒ Dwf' : (g, b : block (
    x : eterm, dx : eof x B[..], a : con, da : cn-of a K[..], at : map a x
  ) ⊢ ewf A[.., x]) }%

invert map-wf' [g, b : block (
    x : eterm, dx : eof x B[..], a : con, da : cn-of a K[..], at : map a x
  ) ⊢ Dwf[.., b.a, b.da]] [_, b ⊢ Dtmap[.., b.a, b.x, b.at]]
% ⇒ Dwf' : (g, x : eterm, dx : eof x B[..] ⊢ ewf A[.., x])
```

Figure 13: Similar theorem statements with different output contextual objects. The statement of `map-wf'` yields the same output's context as found in the original mechanization, whereas the statement of `map-wf` cannot be used in Harpoon without leaving uninstantiated holes to eliminate the unnecessary assumptions `a : con`, `da : cn-of a K[..]` and `at : map a x` from the output's context. The `invert` tactic used on the output of `map-wf'` performs a destructuring of the assumption block resulting from invoking the theorem, and eliminates the assumptions therein that do not depend solely on `x : eterm`. This is because the output judgment `ewf : A[.., x]` only depends on `x` and variables in `g`.

reconstruction errors, as well as uncaught exceptions, which warrants the need for a redesign of the syntax for variable cases. It is also worth noting that parameter variable matching not being implemented in the presence of more than one binder prevents some theorems from being checked.

5 Conclusion

In conclusion, we have illustrated issues with both Beluga and Harpoon in developing proofs with first-class contexts. Beluga's lack of support for existentially-quantified metavariables has resulted in skolemised forms of theorems to be introduced in the mechanization, which reduces its readability and maintainability. We also presented issues with the current implementation of context subsumption in Beluga that prevent Harpoon scripts from being verified, as well as inconsistencies between Harpoon's internal proof syntax and Beluga's which prevents the proof scripts to be translated to Beluga programs. These issues motivate halting progress on translating the mechanization of Standard ML in Beluga until they are resolved.

References

[1] Andrew Cave and Brigitte Pientka. 2018. Mechanizing proofs with logical relations — Kripke-style. *Mathematical Structures in Computer Science* 28, 9 (2018), 1606–1638. https://doi.org/10.1017/S0960129518000154

[2] Karl Crary. 2009. Explicit Contexts in LF (Revised). http://www.cs.cmu.edu/~crary/papers/2009/excon-rev.pdf

[3] Karl Crary. 2009. A Syntactic Account of Singleton Types via Hereditary Substitution. , 21-29 pages. https://doi.org/10.1145/1577824.1577829

[4] Jacob Errington, Junyoung Clare Jang, and Brigitte Pientka. 2020. Mechanizing Meta-Theory Interactively.

[5] Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A Framework for Defining Logics. *J. ACM* 40, 1 (Jan. 1993), 143–184. https://doi.org/10.1145/138027.138060

[6] Robert Harper and Mark Lillibridge. 1994. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 123–137. https://doi.org/10.1145/174675.176927

[7] Robert Harper and Christopher A. Stone. 2000. A Type-Theoretic Interpretation of Standard ML. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. The MIT Press. https://doi.org/10.7551/mitpress/5641.003.0019 arXiv:https://direct.mit.edu/chapter-pdf/186174/9780262281676_cam.pdf

[8] Gerwin Klein and Tobias Nipkow. 2006. A Machine-Checked Model for a Java-like Language, Virtual Machine, and Compiler. *ACM Trans. Program. Lang. Syst.* 28, 4 (July 2006), 619–695. https://doi.org/10.1145/1146809.1146811

[9] Daniel K. Lee, Karl Crary, and Robert Harper. 2007. Towards a Mechanized Metatheory of Standard ML. *SIGPLAN Not.* 42, 1 (Jan. 2007), 173–184. https://doi.org/10.1145/1190215.1190245

[10] Robin Milner, Mads Tofte, and David Macqueen. 1997. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA.

[11] Michael Norrish. 1998. C formalised in HOL.

[12] Scott Owens. 2008. A Sound Semantics for $\text{Caml}_{\text{light}}$. In *Proceedings of the Theory and Practice of Software, 17th European Conference on Programming Languages and Systems* (Budapest, Hungary) *(ESOP'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 1–15.

[13] Frank Pfenning and Carsten Schürmann. 1999. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *Automated Deduction — CADE-16*. Springer Berlin Heidelberg, Berlin, Heidelberg, 202–206.

[14] Frank Pfenning and Carsten Schürmann. 2002. Twelf User's Guide. http://www.cs.cmu.edu/~twelf/guide-1-4/twelf.pdf

[15] Brigitte Pientka. 2001. Termination and Reduction Checking for Higher-Order Logic Programs. In *Proceedings of the First International Joint Conference on Automated Reasoning (IJCAR '01)*. Springer-Verlag, Berlin, Heidelberg, 401–415.

[16] Brigitte Pientka and Joshua Dunfield. 2010. Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description). In *Automated Reasoning*, Jürgen Giesl and Reiner Hähnle (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 15–21. https://doi.org/10.1007/978-3-642-14203-1_2

[17] Brigitte Pientka and Ryan Kavanagh. 2014. A beginner's guide to programming in Beluga. http://complogic.cs.mcgill.ca/tutorial.pdf

[18] Christopher A. Stone and Robert Harper. 2000. Deciding Type Equivalence in a Language with Singleton Kinds. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Boston, MA, USA) *(POPL '00)*. Association for Computing Machinery, New York, NY, USA, 214–227. https://doi.org/10.1145/325694.325724

[19] Christopher A. Stone and Robert Harper. 2000. *Singleton Kinds and Singleton Types*. Ph.D. Dissertation. Carnegie Mellon University, USA. AAI3002766.

[20] Christopher A. Stone and Robert Harper. 2006. Extensional Equivalence and Singleton Types. *ACM Transactions Compututational Logic* 7, 4 (Oct. 2006), 676–722. https://doi.org/10.1145/1183278.1183281

A Syntax and semantics

This appendix features the syntax and semantics of the internal language as presented in the original mechanization [9]. It is not a research contribution, but is included for presentation purposes.

A.1 Internal language syntax

C	::=		constructors:
		α	variable
		$\langle\rangle$	unit constructor
		$\langle \mathrm{C}_1, \mathrm{C}_2 \rangle$	pairs
		$\pi_1 \mathrm{C}$	left projection
		$\pi_2 \mathrm{C}$	right projection
		$\lambda\alpha{:}\mathrm{K}.\mathrm{C}$	abstraction
		$\mathrm{C}_1\ \mathrm{C}_2$	application
		$\mathtt{Unit}$	unit type
		$\mathrm{C}_1 \times \mathrm{C}_2$	products
		$\mathrm{C}_1 \to \mathrm{C}_2$	functions
		$\mathrm{C}_1 + \mathrm{C}_2$	sums
		$\mathtt{Ref}\ \mathrm{C}_1$	references
		$\mathtt{Tag}\ \mathrm{C}_1$	generative tags
		$\mathtt{Tagged}$	tagged expressions
		$\mu\alpha{:}\mathbf{T}.\mathrm{C}$	recursive types
K	::=		kinds:
		1	unit kind
		$\mathbf{T}$	types
		$\mathcal{S}(\mathrm{C})$	singleton kind
		$\Pi\alpha{:}\mathrm{K}.\mathrm{C}$	dependent functions
		$\Sigma\alpha{:}\mathrm{K}.\mathrm{C}$	dependent pairs

ℓ	::=	$\ldots$	locations:
e	::=		terms:
		x	variables
		$\langle\rangle$	unit term
		$\langle e_1, e_2 \rangle$	pairs
		$\pi_1 e$	left projection
		$\pi_2 e$	right projection
		$\mathtt{fun}\ x(y{:}\mathrm{C}_1){:}\mathrm{C}_2.e$	recursive function
		$e_1\ e_2$	application
		$\mathtt{inl}_{\mathrm{C}} e$	left sum intro
		$\mathtt{inr}_{\mathrm{C}} e$	right sum intro
		$\mathtt{case}(e_1, x.e_2, y.e_3)$	case
		$\mathtt{loc}\ \ell$	locations
		$\mathtt{ref}\ e$	new reference
		$!e$	dereference
		$e_1 := e_2$	assignment
		$\mathtt{tagloc}\ \ell$	tag literal
		$\mathtt{newtag}_{\mathrm{C}}$	new tag
		$\mathtt{tag}(e_1, e_2)$	tag injection
		$\mathtt{iftagof}$ $(e_1, e_2, x.e_3, e_4)$	tag check
		$\mathtt{raise}\ e$	raise exception
		$\mathtt{try}(e_1, x.e_2)$	try/handle
		$\mathtt{roll}_{\mathrm{C}} e$	recursive type intro
		$\mathtt{unroll}\ e$	recursive type elim
		$\mathtt{snd}(M)$	module projection

M	::=		modules:
		s	variables
		$\langle\rangle$	unit module
		$[e]$	term module
		$[\mathrm{C}]$	constructor module
		$\langle M_1, M_2 \rangle$	pairs
		$\pi_1 M$	left projection
		$\pi_2 M$	right projection
		$\lambda(s/\alpha_s{:}\sigma_1) :> \alpha_2.M$	functor
		$M_1\ M_2$	application
		$M :> \sigma$	sealing
		$\mathtt{let}\ s/\alpha_s = M_1$ $\mathtt{in}(M_2 :> \sigma)$	let binding

A.2 Internal language static semantics

$\boxed{\Gamma \vdash \mathrm{K}}$ Kind well-formedness

$$\frac{}{\Gamma \vdash 1} \qquad \frac{}{\Gamma \vdash \mathbf{T}} \qquad \frac{\Gamma \vdash \mathrm{C} : \mathbf{T}}{\Gamma \vdash \mathcal{S}(\mathrm{C})}$$

$$\frac{\Gamma \vdash \mathrm{K}' \quad \Gamma, \alpha{:}\mathrm{K}' \vdash \mathrm{K}'' \quad \alpha \notin \mathrm{Dom}(\Gamma)}{\Gamma \vdash \Sigma\alpha{:}\mathrm{K}'.\mathrm{K}''}$$

$$\frac{\Gamma \vdash \mathrm{K}' \quad \Gamma, \alpha{:}\mathrm{K}' \vdash \mathrm{K}'' \quad \alpha \notin \mathrm{Dom}(\Gamma)}{\Gamma \vdash \Pi\alpha{:}\mathrm{K}'.\mathrm{K}''}$$

$\boxed{\Gamma \vdash \mathrm{C} : \mathrm{K}}$ Constructor well-formedness

$$\frac{\alpha{:}\mathrm{K} \in \Gamma}{\Gamma \vdash \alpha : \mathrm{K}} \qquad \frac{s/\alpha_s{:}\sigma \in \Gamma}{\Gamma \vdash \alpha_s : \mathtt{Fst}(\sigma)} \qquad \frac{}{\Gamma \vdash \mathtt{Unit} : \mathbf{T}}$$

$$\frac{}{\Gamma \vdash \mathtt{Tagged} : \mathbf{T}} \qquad \frac{\Gamma \vdash \mathrm{C} : \mathbf{T}}{\Gamma \vdash \mathtt{Ref}\ \mathrm{C} : \mathbf{T}} \qquad \frac{\Gamma \vdash \mathrm{C} : \mathbf{T}}{\Gamma \vdash \mathtt{Tag}\ \mathrm{C} : \mathbf{T}}$$

$$\frac{\Gamma \vdash \mathrm{C}_1 : \mathbf{T} \quad \Gamma \vdash \mathrm{C}_2 : \mathbf{T}}{\Gamma \vdash \mathrm{C}_1 \times \mathrm{C}_2 : \mathbf{T}} \qquad \frac{\Gamma \vdash \mathrm{C}_1 : \mathbf{T} \quad \Gamma \vdash \mathrm{C}_2 : \mathbf{T}}{\Gamma \vdash \mathrm{C}_1 \to \mathrm{C}_2 : \mathbf{T}}$$

$$\frac{\Gamma \vdash \mathrm{C}_1 : \mathbf{T} \quad \Gamma \vdash \mathrm{C}_2 : \mathbf{T}}{\Gamma \vdash \mathrm{C}_1 + \mathrm{C}_2 : \mathbf{T}}$$

$$\frac{\Gamma, \alpha{:}\mathbf{T} \vdash \mathrm{C} : \mathbf{T} \quad \alpha \notin \mathrm{Dom}(\Gamma)}{\Gamma \vdash \mu\alpha{:}\mathbf{T}.\mathrm{C} : \mathbf{T}} \qquad \frac{}{\Gamma \vdash \langle\rangle : 1}$$

$$\frac{\Gamma \vdash \mathrm{C}_1 : \mathrm{K}_1 \quad \Gamma \vdash \mathrm{C}_2 : \mathrm{K}_2}{\Gamma \vdash \langle \mathrm{C}_1, \mathrm{C}_2 \rangle : \mathrm{K}_1 \times \mathrm{K}_2} \qquad \frac{\Gamma \vdash \mathrm{C} : \Sigma\alpha{:}\mathrm{K}'.\mathrm{K}''}{\Gamma \vdash \pi_1\mathrm{C} : \mathrm{K}'}$$

$$\frac{\Gamma \vdash \mathrm{C} : \Sigma\alpha{:}\mathrm{K}'.\mathrm{K}''}{\Gamma \vdash \pi_2\mathrm{C} : [\pi_1\mathrm{C}/\alpha]\mathrm{K}''}$$

$$\frac{\Gamma, \alpha{:}\mathrm{K}' \vdash \mathrm{C} : \mathrm{K}'' \quad \Gamma \vdash \mathrm{K}' \quad \alpha \notin \mathrm{Dom}(\Gamma)}{\Gamma \vdash \lambda\alpha{:}\mathrm{K}'.\mathrm{C} : \Pi\alpha{:}\mathrm{K}'.\mathrm{K}''}$$

$$\frac{\Gamma \vdash \mathrm{C}_1 : \Pi\alpha{:}\mathrm{K}'.\mathrm{K}'' \quad \Gamma \vdash \mathrm{C}_2 : \mathrm{K}'}{\Gamma \vdash \mathrm{C}_1\ \mathrm{C}_2 : [\mathrm{C}_2/\alpha]\mathrm{K}''} \qquad \frac{\Gamma \vdash \mathrm{C} : \mathbf{T}}{\Gamma \vdash \mathrm{C} : \mathcal{S}(\mathrm{C})}$$

$$\frac{\Gamma \vdash \pi_1\mathrm{C} : \mathrm{K}_1 \quad \Gamma \vdash \pi_2\mathrm{C} : \mathrm{K}_2}{\Gamma \vdash \mathrm{C} : \mathrm{K}_1 \times \mathrm{K}_2}$$

$$\frac{\Gamma \vdash \mathrm{C} : \Pi\alpha{:}\mathrm{K}'.\mathrm{L} \quad \Gamma, \alpha{:}\mathrm{K}' \vdash \mathrm{C}\ \alpha : \mathrm{K}'' \quad \alpha \notin \mathrm{Dom}(\Gamma)}{\Gamma \vdash \mathrm{C} : \Pi\alpha{:}\mathrm{K}'.\mathrm{K}''}$$

$$\frac{\Gamma \vdash \mathrm{C} : \mathrm{K}' \quad \Gamma \vdash \mathrm{K}' \leq \mathrm{K}}{\Gamma \vdash \mathrm{C} : \mathrm{K}}$$

$$\frac{\Gamma \vdash \mathrm{C} : \mathrm{K}' \quad \Gamma \vdash \mathrm{K}' \equiv \mathrm{K}}{\Gamma \vdash \mathrm{C} : \mathrm{K}}$$

$\boxed{\Gamma \vdash \mathrm{K}_1 \equiv \mathrm{K}_2}$ Kind equivalence

$$\frac{}{\Gamma \vdash 1 \equiv 1} \qquad \frac{}{\Gamma \vdash \mathbf{T} \equiv \mathbf{T}} \qquad \frac{\Gamma \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathbf{T}}{\Gamma \vdash \mathcal{S}(\mathrm{C}_1) \equiv \mathcal{S}(\mathrm{C}_2)}$$

$$\frac{\Gamma \vdash \mathrm{K}'_1 \equiv \mathrm{K}'_2 \quad \Gamma, \alpha{:}\mathrm{K}'_1 \vdash \mathrm{K}''_1 \equiv \mathrm{K}''_2 \quad \alpha \notin \mathrm{Dom}(\Gamma)}{\Gamma \vdash \Sigma\alpha{:}\mathrm{K}'_1.\mathrm{K}''_1 \equiv \Sigma\alpha{:}\mathrm{K}'_2.\mathrm{K}''_2}$$

$$\frac{\Gamma \vdash \mathrm{K}'_1 \equiv \mathrm{K}'_2 \quad \Gamma, \alpha{:}\mathrm{K}'_1 \vdash \mathrm{K}''_1 \equiv \mathrm{K}''_2 \quad \alpha \notin \mathrm{Dom}(\Gamma)}{\Gamma \vdash \Pi\alpha{:}\mathrm{K}'_1.\mathrm{K}''_1 \equiv \Pi\alpha{:}\mathrm{K}'_2.\mathrm{K}''_2}$$

$\boxed{\Gamma \vdash \mathrm{K}_1 \leq \mathrm{K}_2}$ Subkinding

$$\frac{}{\Gamma \vdash 1 \leq 1} \qquad \frac{}{\Gamma \vdash \mathbf{T} \leq \mathbf{T}} \qquad \frac{\Gamma \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathbf{T}}{\Gamma \vdash \mathcal{S}(\mathrm{C}_1) \leq \mathcal{S}(\mathrm{C}_2)}$$

$$\frac{\Gamma \vdash \mathrm{C} : \mathbf{T}}{\Gamma \vdash \mathcal{S}(\mathrm{C}) \leq \mathbf{T}} \qquad \frac{\Gamma \vdash \mathrm{K}'_1 \leq \mathrm{K}'_2 \quad \Gamma, \alpha{:}\mathrm{K}'_1 \vdash \mathrm{K}''_1 \leq \mathrm{K}''_2 \quad \Gamma, \alpha{:}\mathrm{K}'_2 \vdash \mathrm{K}''_2 \quad \alpha \notin \mathrm{Dom}(\Gamma)}{\Gamma \vdash \Sigma\alpha{:}\mathrm{K}'_1.\mathrm{K}''_1 \leq \Sigma\alpha{:}\mathrm{K}'_2.\mathrm{K}''_2}$$

$$\frac{\Gamma \vdash \mathrm{K}'_2 \leq \mathrm{K}'_1 \quad \Gamma, \alpha{:}\mathrm{K}'_2 \vdash \mathrm{K}''_1 \leq \mathrm{K}''_2 \quad \Gamma, \alpha{:}\mathrm{K}'_1 \vdash \mathrm{K}''_1 \quad \alpha \notin \mathrm{Dom}(\Gamma)}{\Gamma \vdash \Pi\alpha{:}\mathrm{K}'_1.\mathrm{K}''_1 \leq \Pi\alpha{:}\mathrm{K}'_2.\mathrm{K}''_2}$$

B Core lemmas

This appendix presents some of the core lemmas involved in proving type preservation for terms of the internal language [9]. It is not a research contribution, but is included for presentation purposes.

B.1 Validity

Theorem 1 (Functionality for kinds)**.** *If* $\Gamma, \alpha{:}\mathrm{K} \vdash \mathrm{K}'$*, and* $\Gamma \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathrm{K}$*, and* $\Gamma \vdash \mathrm{C}_1 : \mathrm{K}$ *and* $\Gamma \vdash \mathrm{C}_2 : \mathrm{K}$*, then* $\Gamma \vdash [\mathrm{C}_1/\alpha]\mathrm{K}' \equiv [\mathrm{C}_2/\alpha]\mathrm{K}'$*.*

Theorem 2 (Validity for constructors and kinds)**.**

1. *If* $\Gamma \vdash \mathrm{C} : \mathrm{K}$*, then* $\Gamma \vdash \mathrm{K}$*.*
2. *If* $\Gamma \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathrm{K}$*, then* $\Gamma \vdash \mathrm{C}_1 : \mathrm{K}$*, and* $\Gamma \vdash \mathrm{C}_2 : \mathrm{K}$*, and* $\Gamma \vdash \mathrm{K}$*.*
3. *If* $\Gamma \vdash \mathrm{K}_1 \equiv \mathrm{K}_2$*, then* $\Gamma \vdash \mathrm{K}_1$ *and* $\Gamma \vdash \mathrm{K}_2$*.*
4. *If* $\Gamma \vdash \mathrm{K}_1 \leq \mathrm{K}_2$*, then* $\Gamma \vdash \mathrm{K}_1$ *and* $\Gamma \vdash \mathrm{K}_2$*.*

B.2 Injectivity

Theorem 3 (Injectivity of simple product types)**.** *If* $\Gamma \vdash \mathrm{C}_1 \times \mathrm{C}_2 \equiv \mathrm{C}'_1 \times \mathrm{C}'_2 : \mathbf{T}$*, then* $\Gamma \vdash \mathrm{C}_1 \equiv \mathrm{C}'_1 : \mathbf{T}$ *and* $\Gamma \vdash \mathrm{C}_2 \equiv \mathrm{C}'_2 : \mathbf{T}$*.*

Theorem 4 (Injectivity of simple arrow types)**.** *If* $\Gamma \vdash \mathrm{C}_1 \to \mathrm{C}_2 \equiv \mathrm{C}'_1 \to \mathrm{C}'_2 : \mathbf{T}$*, then* $\Gamma \vdash \mathrm{C}_1 \equiv \mathrm{C}'_1 : \mathbf{T}$ *and* $\Gamma \vdash \mathrm{C}_2 \equiv \mathrm{C}'_2 : \mathbf{T}$*.*

Theorem 5 (Injectivity of simple sum types)**.** *If* $\Gamma \vdash \mathrm{C}_1 + \mathrm{C}_2 \equiv \mathrm{C}_1' + \mathrm{C}_2' : \mathbf{T}$*, then* $\Gamma \vdash \mathrm{C}_1 \equiv \mathrm{C}_1' : \mathbf{T}$ *and* $\Gamma \vdash \mathrm{C}_2 \equiv \mathrm{C}_2' : \mathbf{T}$*.*

Theorem 6 (Injectivity of simple reference types)**.** *If* $\Gamma \vdash \mathtt{ref}\ \mathrm{C} \equiv \mathtt{ref}\ \mathrm{C}' : \mathbf{T}$*, then* $\Gamma \vdash \mathrm{C} \equiv \mathrm{C}' : \mathbf{T}$*.*

Theorem 7 (Injectivity of simple tag types)**.** *If* $\Gamma \vdash \mathtt{tag}\ \mathrm{C} \equiv \mathtt{tag}\ \mathrm{C}' : \mathbf{T}$*, then* $\Gamma \vdash \mathrm{C} \equiv \mathrm{C}' : \mathbf{T}$*.*

B.3 Type presevation

Theorem 8 (Monotonicity for terms)**.** *If* $\Gamma; (\Upsilon, \Theta) \vdash e : \mathrm{C}$ *and* $\ell \notin \mathrm{Dom}(\Upsilon)$*, then* $\Gamma; ((\Upsilon, \ell{:}\mathrm{C}'), \Theta) \vdash e : \mathrm{C}$*. Further, if* $\Gamma; (\Upsilon, \Theta) \vdash e : \mathrm{C}$ *and* $\ell \notin \mathrm{Dom}(\Theta)$*, then* $\Gamma; (\Upsilon, (\Theta, \ell{:}\mathrm{C}')) \vdash e : \mathrm{C}$*.*

Theorem 9 (Preservation for terms)**.** *If* $(e, S) \mapsto (e', S')$*, and* $\cdot; \Phi \vdash e : \mathrm{C}$*, and* $\Phi \vdash S : \Phi$*, then there exists some extension* Φ' *of* Φ *such that* $\cdot; \Phi' \vdash e' : \mathrm{C}$ *and* $\Phi' \vdash S' : \Phi'$*.*