FACULTY OF
COMPUTER SCIENCE

Theoretical Computer Science

Master's Thesis

Induction Provers in Hets: Leveraging the Tons of Inductive Problems language and tools to talk to more Automated Theorem Provers

Tom Kranz

Supervisor: Prof. Dr.-Ing. Till Mossakowski
Assisting Supervisor: Dipl.-Inf. Mario Frank

Institute for Intelligent Cooperating Systems, Otto von Guericke University Magdeburg

2022-10-16

To Vera, whom I hope to be able to call a friend for years to come.
You carried me through the hardest parts of this ordeal,
and were the reason I finished a Bachelor's degree in the first place.
Thank you.

Abstract

Abstract

This thesis documents the process of integrating the Tons of Inductive Problems (TIP) format with the Heterogeneous Tool Set (Hets) to simplify communication with automated theorem provers (ATPs) capable of proving inductively. This integration is evaluated by first connecting the inductive ATP Zipperposition to Hets and then comparing the newly gained automatic induction capabilities with the pre-existing capabilities provided by SPASS and E.

Contents

List of Tables

Listings

Abbreviations

ATP automated theorem prover. v, 1, 2, 17, 19–21, 23, 29, 32, 39, 40

CASL the Common Algebraic Specification Language. 6, 7, 9–12, 23, 25–27, 29–32, 35, 36, 39, 40

$CFOL^{=}$ first-order logic with sort generation constraints and equality – the subinstitution of $PCFOL^{=}$ without partial functions. 10, 24–28, 31, 39, 40

CL Common Logic – a knowledge-representation logic based on untyped first-order logic. 2, 40

FOF first-order form – a format of the TPTP for expressing problems in untyped first-order logic. 21

FOL first-order logic. 2, 17, 26, 27, 32, 33, 35, 37

Hets the Heterogeneous Tool Set. v, 3, 5, 19–21, 23, 26–33, 35–37, 39, 40

HOL higher-order logic. 15, 17, 33, 37, 40

$PCFOL^{=}$ first-order logic with partial functions, sort generation constraints, and equality – the subinstitution of $SubPCFOL^{=}$ without subsorting. 10, 11, 23, 24, 26

$PolyTyConsHOL^{=}$ polymorphic higher-order logic with type constructor definitions and equality – the subinstitution of $SubPCoClTyConsProdsHOL^{=}$ without subsorting, partial functions, type constructor class definitions, and interpreted product type constructor. 27

$SubPCFOL^{=}$ subsorted first-order logic with partial functions, sort generation constraints, and equality – the institution encompassing the full expressive power of CASL. 6, 7, 10, 26–28, 31

$SubPCoClTyConsProdsHOL^{=}$ subsorted higher-order logic with partial functions, type constructor classes, type constructor definitions, interpreted product type constructors, and equality – the institution encompassing the full expressive power of HasCASL. 11, 14, 15, 27, 29

TFF typed first-order form – a format of the TPTP for expressing problems in typed first-order logic. 19, 21

THF typed higher-order form – a format of the TPTP for expressing problems in typed higher-order logic. 32

TIP Tons of Inductive Problems – a library of test problems for ATPs, specifically problems requiring induction proofs to solve. v, 16, 17, 19, 20, 23, 26, 27, 29–32, 35, 36, 39, 40

TPTP Thousands of Problems for Theorem Provers – a library of test problems for ATPs covering various problem classes. 19, 32, 40

CHAPTER 1

Introduction

> Using a mathematically formal approach to software development was proposed at an early stage in the development of computer science. [...] It quickly became clear that manual proofs could only be developed for very small systems. Program proving is now supported by large-scale automated theorem proving software, which has meant that larger systems can be proved. However, developing the proof obligations for theorem provers is a difficult and specialized task, so formal verification is not widely used. [75, p. 300]

This excerpt from arguably one of the classic introductory textbooks on software engineering illustrates two of the hurdles formal methods have to take on their way from the ivory towers of theoretical computer science departments into "real-world" software development: Finding proofs that software does what it is supposed to do and specifying the desired behaviour of software in a way that permits such reasoning in the first place. Sommerville goes on to elaborate on reasons for the difficulty of the latter hurdle: "Problem owners and domain experts cannot understand a formal specification, so they cannot check that it accurately represents their requirements." [75, p. 302] One means of narrowing this semantic gap are formalisms that incorporate domain-specific concepts in their design, thus enabling domain experts to contribute to formal specifications from a perspective they are already familiar with.

When such a formalism is first introduced, it may raise the first hurdle mentioned above because existing automated theorem provers (ATPs) will likely not know how to handle it. One needs to establish new automation tools or find ways to translate new specifications into existing formalisms with the desired automation support or just limit the new formalism's applications to small use-cases that can reasonably be proven manually. In large software systems, certain parts may be specified with their own formalisms, but still need to interact with each other. This need for interoperability between specifications suggests that translations between the involved formalisms or into a common denominator between them may even be required.

The conditions for, methods to realize, and consequences of such interoperation are the subject of the study of structured and heterogeneous specification. One way to make specifications sufficiently abstract for general study is to model them algebraically: If software systems are considered descriptions of algebras, i.e. structures consisting of a set of data and a set of functions on the data, specifications can be considered definitions of entire classes of

algebras. Such a definition in turn extends to the software systems realizing a specification via membership of their algebra in the specified class. [69, pp. 3, 5] Structured specification, then, studies operations for combining specifications and the effects of these operations on the defined classes of algebras, while heterogeneous specification studies operations arising between algebras of differing internal structure.

An example of a formalism that was created specifically to relieve its intended users from the idiosyncracies and limitations of a prevalent standard is the Common Logic (CL) framework [81]. CL is a logic based on first-order logic (FOL) but extended to make knowledge representation in the context of the Semantic Web more natural than in FOL. [46] In an effort to retain existing tool support, it was designed to permit different levels of expressiveness through syntactic restrictions that should suffice, as Menzel puts it, "for the vast majority of practical purposes". For example, CL provides sequence variables to express lists as arguments for functions and predicates. As lists are inductively defined data structures, they require an induction scheme to fully specify in FOL, making CL non-compact and therefore impossible to develop a complete ATP for. [34, 66] Allowing sequence variables to only be introduced at the outermost universal-quantification level of an axiom, though, alleviates this problem by enabling a reduction of the axiom to an infinite set of sequence-variable-free sentences for which compactness still holds. If maximum ATP support is desired, it makes sense to consider CL specifications that violate this restriction separately from those that satisfy it, using incomplete provers only for the former and complete FOL provers for the latter. Proving soundness for the combination of realizations of these specifications may then degenerate into a heterogeneous task, since they may come from differently-expressive programming languages, each with its own notion of algebraic models.

As foreshadowed above, a non-compact logic of particular practical interest is the extension of FOL by an induction scheme, since many "everyday" structures in computer science are defined inductively: natural numbers[1], lists, trees, to name just a few prominent ones. Of course, specifications defining and using such completeness-breaking structures should nonetheless be treated with the best automation support possible. It is therefore hardly surprising that, despite their predetermined incompleteness, ATPs for non-compact logics still emerge, especially those "just" adding induction capabilities to existing FOL proof calculi. [83, 24]

The remainder of this work will detail the steps I have taken to augment a tool for heterogeneous proof management by an interface for communication with induction-capable ATPs. Chapter 2 introduces the tool, its theoretical foundation and the facilities used to establish this interface. Chapter 3 collects existing uses of ATPs in interactive tools to establish a context for this work's contribution. Chapter 4 details this contribution by describing the changes to the tool's code and explaining the experiments by which the usefulness of these changes has been measured. Chapter 5 presents the results of these experiments and evaluates whether these can be deemed a success and which factors played a role in that outcome. Chapter 6 summarizes the knowledge gained through this work and gives pointers towards an expansion of that knowledge.

[1] Usually demoted to finite subsets whose members are more easily digestible for real processors – but results for the entirety of $\mathbb{N}$ obviously ease reasoning within these subsets.

CHAPTER 2

Background

This chapter will give the required theoretical background for the work that I will do. I will try to include everything that is needed to comprehend the decisions I will make. What will not be introduced here is some basic category theoretical vocabulary, which the reader should have heard before but need not be intimately familiar with. Recommending a textbook would greatly overstate the extent to which category theoretical insights will be required, so I will simply refer the reader to their favourite encyclopedia for any such vocabulary that does not immediately ring a bell.

2.1 The Heterogeneous Tool Set

The Heterogeneous Tool Set (Hets) is a proof management and formal methods integration software. It implements the heterogeneous specification framework described by Mossakowski. [51, 52] This includes a parser, static analysis and a proof engine for heterogeneous multi-logic specifications as well as connections to provers and model finders for the individual logics involved in these specifications. The semantics of these heterogeneous specifications are defined by the theory of institutions [35].

2.1.1 Institutions and specifications

An institution $I = (\mathbf{Sign}, \mathrm{Sen}, \mathrm{Mod}, \models)$ formalizes a logic by defining:

- a category **Sign** with signatures as objects and signature translations as morphisms,
- a covariant functor $\mathrm{Sen}\colon \mathbf{Sign} \to \mathbf{Set}$ assigning each signature $\Sigma \in |\mathbf{Sign}|$ the language of logic sentences using non-logic symbols from Σ and each signature translation $\sigma \in \hom(\mathbf{Sign}^I)$ a corresponding sentence translation along σ,
- a contravariant functor $\mathrm{Mod}\colon \mathbf{Sign}^{\mathbf{op}} \to \mathbf{Cat}$ assigning each signature Σ a category of models whose objects interpret the objects of Σ and may be connected through morphisms among each other, and each $\sigma\colon \Sigma \to \Sigma'$ a reduct functor against σ, i.e. $\mathrm{Mod}(\sigma)\colon \mathrm{Mod}(\Sigma') \to \mathrm{Mod}(\Sigma)$, and
- a family of satisfaction relations $\models$, that, for each signature Σ, relates Σ-models with the Σ-sentences they satisfy, i.e. $\models_\Sigma \subseteq |\mathrm{Mod}(\Sigma)| \times \mathrm{Sen}(\Sigma)$,

such that for each signature translation $\sigma\colon \Sigma \to \Sigma'$ of **Sign** the satisfaction condition holds:

$$\forall M' \in |\mathrm{Mod}(\Sigma')|.\, \forall \varphi \in \mathrm{Sen}(\Sigma).\, M' \models_{\Sigma'} \sigma(\varphi) \iff M'|_\sigma \models_\Sigma \varphi,$$

where $M'|_\sigma := \mathrm{Mod}(\sigma)(M')$ and $\sigma(\varphi) := \mathrm{Sen}(\sigma)(\varphi)$. Application of the reduct functor to model morphisms may also be abbreviated: $h|_\sigma := \mathrm{Mod}(\sigma)(h)$. The satisfaction relation can be extended to sets of sentences $\Gamma \subseteq \mathrm{Sen}(\Sigma)$, i.e. $M \models_\Sigma \Gamma \iff \forall \varphi \in \Gamma.\, M \models_\Sigma \varphi$. It can also be used to relate sets of sentences with single sentences, which is referred to as semantic entailment: $\Gamma \models_\Sigma \varphi \iff \forall M \in |\mathrm{Mod}(\Sigma)|.\, M \models_\Sigma \Gamma \implies M \models_\Sigma \varphi$.

A theory over I is a tuple $T = (\Sigma, \Gamma)$ with Σ again a signature and $\Gamma \subseteq \mathrm{Sen}(\Sigma)$ its set of axioms. A theory by itself already determines a class – and by extension a full subcategory – of models that satisfy its axioms: $|\mathrm{Mod}(T)| = \{M \in |\mathrm{Mod}(\Sigma)| \mid M \models_\Sigma \Gamma\}$. It can thus be considered a specification and indeed, so called flat specifications are just theories with finite sets of axioms. Structured specifications, on the other hand, combine existing specifications using specification-building operations such as:

union: For a signature Σ and Σ-specifications SP_1, SP_2,
$|\mathrm{Mod}(SP_1 \cup SP_2)| = |\mathrm{Mod}(SP_1)| \cap |\mathrm{Mod}(SP_2)|$.

translation: For a signature morphism $\sigma\colon \Sigma \to \Sigma'$ and a Σ-specification SP,
$|\mathrm{Mod}(\text{translate } SP \text{ by } \sigma)| = \{M' \in |\mathrm{Mod}(\Sigma')| \mid M'|_\sigma \in |\mathrm{Mod}(SP)|\}$

hiding: For a signature morphism $\sigma\colon \Sigma \to \Sigma'$ and a Σ'-specification SP',
$|\mathrm{Mod}(\text{derive from } SP' \text{ by } \sigma)| = \{M'|_\sigma \mid M' \in |\mathrm{Mod}(SP')|\}$

free extension: For a signature morphism $\sigma\colon \Sigma \to \Sigma'$ and a Σ'-specification SP',
$|\mathrm{Mod}(\text{free } SP' \text{ along } \sigma)| = \{M' \in |\mathrm{Mod}(SP')| \mid$
M' is strongly persistently $(\mathrm{Mod}(\sigma)\colon \mathrm{Mod}(SP') \to \mathrm{Mod}(\Sigma))$-free$\}$,
i.e. for any model $N' \in |\mathrm{Mod}(SP')|$ and any model morphism $h\colon M'|_\sigma \to N'|_\sigma$, there is a unique morphism $h^\#\colon M' \to N'$ such that $h^\#|_\sigma = h$. [50, Sec. 2.3] In an algebraic context and with σ an embedding, such a model M' is an absolutely free Σ-structure with generators in $M'|_\sigma$ [3, Def. 7]. Roughly speaking, all the objects of M' have to be reachable by applying functions of M' to already reachable objects, with the objects in $M'|_\sigma$ being considered reachable to begin with.

Structured specifications containing only unions, translations, and flat specifications are flattenable into a normal form, i.e. a flat specification with the same model class. In an institution with the so-called weak amalgability property, hiding specifications can also be brought into normal form. Free extensions cannot generally be brought into normal form, since they allow the specification of inductive types and relations and can therefore not be expressed in flat specifications of a compact institution. A specification SP_1 is said to refine another one SP_2 if $|\mathrm{Mod}(SP_1)| \subseteq |\mathrm{Mod}(SP_2)|$.

An institution only captures the model-theoretic view on a logic. To capture proof theory, an institution might be extended by an entailment system to form a logic $L = (\mathbf{Sign}, \mathrm{Sen}, \mathrm{Mod}, \models, \vdash)$. For any signature Σ, the relation $\vdash_\Sigma$ in the entailment system has to be Tarskian, that means for any Σ-sentence φ and sets of Σ-sentences Γ, Γ' it has to be

reflexive: $\{\varphi\} \vdash_\Sigma \varphi$

monotonic: $\Gamma \vdash_\Sigma \varphi \wedge \Gamma' \supseteq \Gamma \implies \Gamma' \vdash_\Sigma \varphi$

transitive: $\Gamma \cup \Gamma' \vdash_\Sigma \varphi \wedge (\forall \varphi' \in \Gamma'.\, \Gamma \vdash_\Sigma \varphi') \implies \Gamma \vdash_\Sigma \varphi$

Furthermore, the relations must be compatible with signature translations $\sigma\colon \Sigma \to \Sigma'$: $\Gamma \vdash_\Sigma \varphi \implies \sigma(\Gamma) \vdash_\Sigma \sigma(\varphi)$. Most importantly, they have to be sound: $\Gamma \vdash_\Sigma \varphi \implies \Gamma \models_\Sigma \varphi$. A logic where the soundness condition is an equivalence is called complete. Although any institution can easily be extended to a complete logic by defining semantic entailment as the entailment system, this would be missing the point. Rather, an entailment system should be defined via a system of finitary derivation rules, or calculus, which can be used to construct a proof.

For example, Hets implements a logic-independent proof calculus for structured specifications based on development graphs. Logic-specific proof calculi, or external tools implementing them, are only needed upon reaching flat specifications. Heterogeneity is then introduced by viewing institutions as objects of a category with morphisms defined to suit a particular notion of "logic translation"[49].

2.1.2 Institution comorphisms (plain maps)

Goguen and Burstall defined their classical institution morphisms from the point of view of translations as projections from one institution onto another. That is, the source can be considered as being "built on" the target. Such morphisms would allow re-use of proof tools built for the more expressive institution, the source, for specifications of the less expressive one, the target and for target sentences to appear in source specifications. This was achieved by reducing source signatures and models to the target institution and injecting target sentences into the source institution.

A somewhat dual notion to projections are encodings, which are thus formalized under the name of institution *co*morphisms. These were described by Meseguer as plain maps of institutions in order to formalize the notion of a subinstitution in terms of special plain maps. [47] The idea was to translate signatures, signature translations and sentences of a subinstitution I one-to-one to those of the target institution I' and being able to translate models of translated sentences back isomorphically. In the general case though, these maps embody the notion of encoding or representing logical frameworks into another. An institution comorphism $\rho = (\Phi, \alpha, \beta)\colon I \to I'$ thus consists of:

- a functor $\Phi\colon \mathbf{Sign} \to \mathbf{Sign}'$,
- a natural transformation $\alpha\colon \mathrm{Sen} \Rightarrow \mathrm{Sen}' \circ \Phi$, and
- a natural transformation $\beta\colon \mathrm{Mod}' \circ \Phi^{\mathbf{op}} \Rightarrow \mathrm{Mod}$,

such that for each $\Sigma \in |\mathbf{Sign}|$ the satisfaction condition holds:

$$\forall M' \in |\mathrm{Mod}'(\Phi(\Sigma))|.\, \forall \varphi \in \mathrm{Sen}(\Sigma).\, M' \models'_{\Phi(\Sigma)} \alpha_\Sigma(\varphi') \iff \beta_\Sigma(M') \models_\Sigma \varphi.$$

Certain properties of comorphisms allow the borrowing of proof calculi of logics belonging to the target institution, even if the source institution does not have a native logic. Subinstitution comorphisms allow borrowing for structured specifications containing all of the specification-building operations mentioned above. Strongly persistently liberal comorphisms allow borrowing for structured specifications involving specification-building operations using certain signature translations. Such comorphisms can produce a covariant model translation γ_Σ for every (contravariant) β_Σ such that $\beta_\Sigma \circ \gamma_\Sigma = id$. The transformation γ need not be natural but for a signature translation $\sigma\colon \Sigma_1 \to \Sigma_2$ in a specification-building

operation to not disturb borrowing, γ has to be σ-natural: $\gamma_{\Sigma_1} \circ \mathrm{Mod}(\sigma) = \mathrm{Mod}'(\Phi(\sigma)) \circ \gamma_{\Sigma_2}$. Model-expansive comorphisms allow borrowing only for flat specifications.

Theoroidal comorphisms generalize comorphisms by allowing Φ to map signatures in I to theories in I' and signature translations to theory translations.

2.2 The Common Algebraic Specification Language

The Common Algebraic Specification Language (CASL) is a language for the formal specification of functional requirements and modular design of software. [2] Its featureset can be divided into the following layers: [53]

basic specifications for writing (flat) specifications in a particular institution, which defaults to CASL's underlying institution,

structured specifications for building complex specifications from simpler ones,

architectural specifications to prescribe the modular structure of an implementation and enforce the separate development of implementation units, and

libraries of specifications to store and retrieve named specifications.

The CASL institution can be described as subsorted first-order logic with partial functions, sort generation constraints, and equality ($SubPCFOL^{=}$). A sort generation constraint is equivalent to an induction axiom (schema) and so their inclusion in the language makes CASL as a whole non-compact. Basic specifications are built from the following language constructs:

- `sort`, `sorts` for the declaration of one or more sorts or the declaration or definition of one or more subsorts or any combination of these,
- `op`(`s`) for the declaration or definition of one or more constants, total functions and partial functions, optionally with attributes declaring them to be associative, commutative or to have a unit,
- `pred`(`s`) for the declaration or definition of one or more predicates,
- `type`(`s`) to declare one or more sorts with associated constructors and optional selectors without imposing any further restrictions, apart from the expected relationship between selectors and constructors,
- `generated type`(`s`) to declare datatypes as before but with an additional sort generation (no-junk) constraint which ensures that every inhabitant of the declared sort can be generated from the constructors,
- `generated {…}` to subject a group of (sub)sort and operation declarations to a sort generation constraint, i.e. designate the operations in the constraint whose result sorts are also in the constraint as their respective result sort's constructors,
- `free type`(`s`) to declare generated datatypes as before[1] but with an additional no-confusion constraint which ensures that every inhabitant of the declared sort can be uniquely generated from the constructors,

[1] However, the `free {…}` construct is defined in terms of free extensions, which makes it a specification-building operation!

- `var(s)`, `forall` (synonymous) to declare universally quantified global variables of a certain (sub)sort for use in stand-alone axioms or a subsequent `.`-(full-stop-)bulleted list of axioms[2], and
- `axiom(s)` to introduce such axioms using first-order sentences.

A basic specification determines a signature and a set of sentences, i.e. a flat specification over $SubPCFOL^{=}$. In this institution, signatures are 5-tuples $\Sigma = (S, TF, PF, P, \leqslant)$ consisting of the following components:

- S is a set of sort symbols,
- TF and PF are disjoint sets of total and partial function symbols, respectively, associated with a profile, i.e. a non-empty string of sort symbols, the last of which being the result sort and the rest the argument sorts,
- P is a set of predicate symbols associated with a, possibly empty, profile of argument sorts, and
- $\leqslant \subseteq S \times S$ is the subsort relation, a pre-order.

For an example of a CASL basic specification, see an adapted version of the Hets-lib[3] specification of the naturals (`Nat`), positive naturals (`Pos`), and integers (`Int`) and some related operations in Listing 2.1. Lines 1 to 18 induce the below components of the signature Σ_{Num}.

$$
\begin{aligned}
S_{Num} &= \{\texttt{Nat}, \texttt{Int}, \texttt{Pos}\},\\
TF_{Num} &= \{\texttt{0:Nat}, \texttt{suc:NatNat}, \texttt{\_\_-\_\_:NatNatInt},\\
&\quad\ \texttt{1:Nat}, \texttt{\_\_+\_\_:NatNatNat}, \texttt{\_\_-!\_\_:NatNatNat},\\
&\quad\ \texttt{suc:NatPos}, \texttt{1:Pos}, \texttt{\_\_+\_\_:PosNatPos}, \texttt{\_\_+\_\_:NatPosPos},\\
&\quad\ \texttt{\_\_+\_\_:IntIntInt}, \texttt{min:IntIntInt}, \texttt{max:IntIntInt},\\
&\quad\ \texttt{\_\_-\_\_:IntIntInt}, \texttt{-\_\_:IntInt}, \texttt{sign:IntInt}, \texttt{abs:IntNat}\},\\
PF_{Num} &= \{\texttt{pre:NatNat}, \texttt{\_\_-?\_\_:NatNatNat}\},\\
P_{Num} &= \{\texttt{\_\_<=\_\_:NatNat}, \texttt{\_\_>=\_\_:NatNat}, \texttt{even:Nat}, \texttt{odd:Nat},\\
&\quad\ \texttt{\_\_<=\_\_:IntInt}, \texttt{\_\_>=\_\_:IntInt}, \texttt{even:Int}, \texttt{odd:Int}\},\\
\leqslant_{Num} &= \{(\texttt{Nat}, \texttt{Nat}), (\texttt{Pos}, \texttt{Pos}), (\texttt{Int}, \texttt{Int}),\\
&\quad\ (\texttt{Pos}, \texttt{Nat}), (\texttt{Nat}, \texttt{Int}), (\texttt{Pos}, \texttt{Int})\}
\end{aligned}
$$

Lines 1 to 2 additionally produce sort generation constraints for the `Nat` and `Int` and no-confusion axioms for `0:Nat` and `suc:Nat->Nat`. Lines 20 to 53 define the operations and predicates via axioms describing their behaviour. Note that lines 19 to 21 contain attributes for already-declared operations that are expanded to corresponding axioms. These constraints and axioms form the specification's set of sentences Γ_{Num}. The `%(…)%` annotations assign names to sentences that can be used by tools to improve their presentation. The `%implied` annotations in lines 19 to 21 and 50 to 52 establish a claim *about* the specification that the annotated sentences could be removed from Γ_{Num} without changing the

[2] In any case, each variable is quantified independently for every axiom.
[3] `https://github.com/spechub/Hets-lib`, specifically the file `Basic/Numbers.casl`

```
free type Nat ::= 0 | suc(pre:? Nat)
generated type Int ::= __ - __(Nat;Nat)
sort  Nat < Int
preds __ <= __, __ >= __:              Nat * Nat;
      even, odd:                       Nat;
      __ <= __, __ >= __:              Int * Int;
      even, odd:                       Int
op    1: Nat = suc (0)                 %(1_def_Nat)%
sort  Pos = { p: Nat . p >= 1 }        %(Pos_def)%
ops   __ + __, __ -!__ :               Nat * Nat ->  Nat;
      __ -?__ :                        Nat * Nat ->? Nat;
      suc:                             Nat -> Pos;
      1: Pos = suc(0);                 %(1_as_Pos_def)%
      __ + __:                         Pos * Nat -> Pos;
      __ + __:                         Nat * Pos -> Pos;
      __ + __, min, max, __ - __ :     Int * Int -> Int;
      - __, sign:                      Int -> Int;
      abs:                             Int -> Nat;
      __+__: Int * Int -> Int, comm, assoc, unit 0;     %implied
      min:   Int * Int -> Int, comm, assoc;             %implied
      max:   Int * Int -> Int, comm, assoc, unit 0      %implied
forall m,n,r,s : Nat, i,j : Int
   . i <= j <=> i - j in Nat           %(leq_def_Int)%
   . i >= j <=> i <= j                 %(geq_def_Int)%
   . 0 <= n                            %(leq_def1_Nat)%
   . not suc(n) <= 0                   %(leq_def2_Nat)%
   . suc(m) <= suc(n) <=> m <= n       %(leq_def3_Nat)%
   . even(0)                           %(even_0_Nat)%
   . even(suc(m)) <=> odd(m)           %(even_suc_Nat)%
   . odd(m) <=> not even(m)            %(odd_def_Nat)%
   . even(i) <=> even(abs(i))          %(even_def_Int)%
   . odd(i)  <=> not even(i)           %(odd_def_Int)%
   . odd(i)  <=> odd(abs(i))           %(odd_alt_Int)%
   . 0 + m = m                         %(add_0_Nat)%
   . suc(n) + m = suc(n + m)           %(add_suc_Nat)%
   . (m - n) + (r - s) =(m + r) - (n + s) %(add_def_Int)%
   . min(i,j) = i when i <= j else j   %(min_def_Int)%
   . max(i,j) = j when i <= j else i   %(max_def_Int)%
   . n -! m = 0      if m >= suc(n)    %(subTotal_def1_Nat)%
   . n -! m = n -? m if m <= n         %(subTotal_def2_Nat)%
   . m -? n = r <=> m = r + n          %(sub_def_Nat)%
   .  i - j = i + ( - j )              %(sub_def_Int)%
   . - (m - n) = n - m                 %(neg_def_Int)%
   . sign(i) = 0 when i = 0
         else (1 when i >= 1 else -1) %(sign_def_Int)%
   . abs(i) =  i when i >= 0
         else -i                       %(abs_def_Int)%
   . m - n = r - s <=> m + s = r + n   %(equality_Int)%
   . m = m - 0                         %(Nat2Int_embedding)%
   . m >= n <=> m <= n                 %(geq_def_Nat)%   %implied
   . min(m,0)=0                        %(min_0)%         %implied
   . def(m-?n) <=> m >= n              %(sub_dom_Nat)%   %implied
```

Listing 2.1: CASL basic specification for $\mathbb{N}^+ \leqslant \mathbb{N} \leqslant \mathbb{Z}$ and some operations, demonstrating $SubPCFOL^=$ features

overall specification's model class, i.e. that they are implied by the rest of the theory. Structured specifications permit even more semantic annotations that establish claims about the relationship between specifications. Tools can extract such claims as proof obligations. [55, Sec. 5.2.5]

In general, a sentence over a signature Σ is either a closed first-order formula or a sort generation constraint $(\tilde{S}, \tilde{F}, \theta)$. Here, $\theta\colon \bar{\Sigma} \to \Sigma$ is a signature translation originating in some CASL signature $\bar{\Sigma} = (\bar{S}, \bar{TF}, \bar{PF}, \bar{P}, \bar{\leqslant})$ with $\tilde{S} \subseteq \bar{S}$ and $\tilde{F} \subseteq \bar{TF} \cup \bar{PF}$. This is satisfied by Σ-models whose θ-reducts interpret the sorts in $\tilde{S}$ with sets generated by interpretations of operations in $\tilde{F}$. In other words, $M \models_\Sigma (\tilde{S}, \tilde{F}, \theta)$ iff, in $M|_\theta$, every member of a carrier of a sort in $\tilde{S}$ is the denotation of a $\bar{\Sigma}$-term constructed only from operation symbols in $\tilde{F}$ and variables of sorts in $\bar{S} \backslash \tilde{S}$ under some valuation of these variables. First-order formulas can contain definedness, existential equality and strong equality as interpreted predicates.

Finally, CASL models for a signature $(S, TF, PF, P, \leqslant)$ are standard many-sorted models with non-empty carriers for each sort in S and total functions, partial functions and predicates for the symbols of TF, PF, and P with domains and, where applicable, codomains corresponding to the symbols' profiles. Each subsort relation $s \leqslant s'$ further restricts the admissible models to those providing:

- an injective embedding of the carrier of s into that of s' that is compatible with the behaviour of overloaded functions and predicates, and which must be the identity in case $s = s'$,
- a projection that is injective where defined and left-inverse[4] to the embedding, and
- an s-membership predicate that holds only where the projection is defined.

2.2.1 Structuring constructs

For this work, only three of the many CASL structuring constructs are noteworthy:

Extensions

$$SP_1 \texttt{ then } SP_2$$

Extensions in CASL are based on the union operation. The difference is that the signatures of SP_1 and SP_2 need not be the same. In fact, SP_2 need not even determine a full signature by itself because it inherits all the symbols from SP_1 before it adds its own. The union then treats SP_1 also as a specification with the extended signature. The `then` operator is read left-associatively if chained.

Specification definitions

$$\begin{array}{l}\texttt{spec } SN \texttt{ [}SP_1\texttt{] … [}SP_n\texttt{] given } SP_1'', \ldots, SP_m'' \texttt{ =} \\ \quad SP \\ \texttt{end}\end{array}$$

[4] W.r.t. function composition $\circ$ read as "after".

A specification definition assigns the name SN to the extension of the union of the imports SP_i'' by the union of the parameters SP_j by the definition body SP:

$$\texttt{\{ } SP_1'' \texttt{ and … and } SP_m'' \texttt{ \} then \{ } SP_1 \texttt{ and … and } SP_n \texttt{ \} then } SP$$

Examples of specification definitions, including ones without parameters or imports, can be found in Appendix A.1.

Free specifications

$$\texttt{free \{ } SP \texttt{ \}}$$

If a free specification extends a surrounding specification SP_0 with signature Σ_0 and the extended signature is Σ', the result of the extension is an application of the free extension operation: free (SP_0 `then` SP) along σ, where $\sigma\colon \Sigma_0 \to \Sigma'$ is the embedding of the unextended into the extended signature. Otherwise, the free specification restricts the class of models of SP to initial ones, i.e. absolutely free structures without generators.

A basic specification containing just a free datatype declaration `free types` $DD_1\texttt{;…;}DD_n\texttt{;}$ usually determines the same model class as the free specification containing just the datatype declaration: `free { types` $DD_1\texttt{;…;}DD_n\texttt{; }\}$. In the absence of subsorting, the only conditions for this equivalence are that the free datatype declaration does not, through extension, declare an existing sort as freely generated and that any total selector occuring in one of the DD_i has to be used in every constructor of DD_i.

2.2.2 $PCFOL^=$ and subsorting encoding

First-order logic with partial functions, sort generation constraints, and equality ($PCFOL^=$) is the subinstitution of $SubPCFOL^=$ without support for subsorts. That is, any $PCFOL^=$ specification can be trivially mapped to a $SubPCFOL^=$ specification by taking the equality on S as $\leqslant$. CASL basic specifications can be restricted to be $PCFOL^=$ by disallowing constructs such as lines 3 and 9 in Listing 2.1. $SubPCFOL^=$ can be encoded in $PCFOL^=$ by means of a theoroidal subinstitution comorphism that axiomatizes the subsorting requirements mentioned above in the theory when translating a signature with subsorts. Sentences are translated by making the involved injections, projections, and membership predicates explicit. This comorphism is called ($3'$) by Mossakowski. [50]

2.2.3 $CFOL^=$ and partiality encoding

First-order logic with sort generation constraints and equality ($CFOL^=$) is the subinstitution of $PCFOL^=$ without support for partial functions, definedness, and the distinction between existential and strong equality. That is, any $CFOL^=$ specification can be trivially mapped to a $PCFOL^=$ specification by taking $\emptyset$ as PF. There are multiple encodings for partiality features into $CFOL^=$, but the one that works without restricting the allowed sort generation constraints is not a subinstitution comorphism, but merely model-expansive[5], It is called ($4a'$) by Mossakowski. [50] Signature translation works by adding a constant $\bot_s$ and an explicit definedness predicate D_s for each sort s, and axiomatizing that

[5] Actually, model-bijective, but that is not needed for my purposes.

- $\forall x \in s.\, D_s(x) \iff x \neq \bot_s$ for each sort s,
- $\exists x \in s.\, D_s(x)$ for each sort s,
- functions reflect definedness,
- total functions additionally preserve definedness, and
- satisfaction of a predicate implies definedness of its arguments.

The second set of axioms is necessary because the carriers for sorts of the $PCFOL^=$ specification cannot be empty and that requirement is already met with the constants $\bot_s$. Sentences are translated by

- replacing the interpreted definedness predicates with the explicit new ones,
- replacing strong equations with implications of the equation by definedness of at least one of the comparands,
- replacing existential equations by conjunctions of the equation with definedness of one of the comparands,
- relativizing universally quantified sentences by adding definedness of the quantified variable as a premise to the matrix,
- relativizing existentially quantified sentences by adding definedness of the quantified variable as a conjunct to the matrix, and
- adding the $\bot$'s to the sets of constructors in sort generation constraints.

If a specification only contains sort generation constraints with total functions as constructors, this encoding can be made strongly persistently liberal by weakening the first axiom above to a statement of definedness not holding on $\bot$. This removes the restriction that $\bot$ has to be the only undefined element in each sort. For example, this would allow distinct successors of undefined natural numbers, and thus preserve disjointness of the successor function's range with that of $\bot$, which was made a constructor by the sentence translation. The strongly persistently liberal version is called $(4')$. The covariant model translation $\gamma^{(4')}$ adds to each sort's carrier the term $\bot$, and distinct terms for each value of a function application on a tuple outside the function's domain. It adds an interpretation for the definedness predicate as the existing carrier of the corresponding sort. Each function is re-interpreted to yield the corresponding undefined term when applied to a tuple outside its original domain, and its original value else. Predicates are interpreted with their original sets, i.e. they never hold on the new individuals added by $\gamma^{(4')}$.

2.2.4 HasCASL

HasCASL is an extension of CASL with (partial) function types, polymorphism and type constructors. [71] The name derives from its ability to express an executable sublanguage similar to Haskell [41], enabling implementation and rapid prototyping within a specification. This addresses the need for a close tie between specification and implementation language identified for example in Swierstra's experience report [80]. Its expressive power allows it to be used as an intermediary language to support even such informal design processes as CAD/CAM workflows. [42]

The HasCASL institution can be described as a subsorted higher-order logic with partial

functions, type constructor classes, type constructor definitions, interpreted product type constructors, and equality ($SubPCoClTyConsProdsHOL^{=}$). The syntax of HasCASL builds on that of CASL by adding language constructs to express the additional features of its institution: [72]

- `class`(`es`) to declare type classes, possibly as subclasses of one or more kinds. Classes are regarded as subsets of the set of all types, which is itself represented by the built-in type class `Type`. A kind is either a class or a constructor kind Kd_1`->`Kd_2 where Kd_1 is a kind, possibly prepended with `+` or `-`, and Kd_2 is a kind. Sentences may be universally quantified over type classes at the outermost quantification level.
- `class` … `{…}` to declare a type class together with axioms that instance types of the class must satisfy, for example the existence of certain operations on the type and associated properties.
- `var` and `forall` quantify over sort terms and classes in HasCASL. Sort terms extend the sorts of CASL by allowing type constructors and quantification over classes.
- `type` declares type constructors in HasCASL by using previously declared type variables as arguments for both type and instance constructors. Type constructor declarations without instance constructors may be abbreviated by omitting type arguments and assigning it a kind. Type constructors of constructor kind can be be covariant, which means that their application preserves the subtype relation, contravariant, which means that their application reverses the subtype relation, or non-variant. Co- or contravariance are indicated by prepending `+` or `-`, respectively, to the argument kind in a type constructor's kind. The nullary type constructor `Unit` and binary type constructors `*`, `->`, and `->?` are built-in[6] and can be thought of as declared like this, using underscores to indicate infix use:

  unit type (constructor) `type Unit : Type` or simply `type Unit`

  product type constructor `type __*__ : +Type -> +Type -> Type`

  function type constructors `types __->__, __->?__ : -Type -> +Type -> Type`

  This means that given a declaration `sorts a < b; c < d`, the type `a * c` is a subtype of `b * d` and the type `b -> c` is a subtype of `a -> d`.
- `type instance` to declare a type constructor that uses some type classes and postulate that instantiations of the axioms attached to the result class with the result type follow from instantiations of the axioms for the input classes with the input types.
- `class instance` to declare a subclass and postulate the above type instance property for all its instances.

HasCASL signatures are 6-tuples $\Sigma = (C, \leq_C, T, A, O, \leq)$ consisting of the following components:

- C is a set of classes, each associated with a raw kind, i.e. the shape of the class's constituent type constructors' kinds in terms of `Type`, arrows and variance annotations.
- $\leq_C \subseteq C \times K$, where K is the set of kinds over C, is the (explicit) subclass relation.

[6] Nullary type constructors correspond to the regular CASL types, while the built-in binary type constructors generalize CASL's operator signatures.

```
1  var a, b, c: Type
2  free type List a ::= nil | cons a (List a)
3  ops map: (a ->? b) -> List a ->? List b;
4       all: Pred a -> List a -> Logical;
5       filter: List a -> Pred a -> List a
6  var x: a; l: List a; f: a ->? b; g: b ->? c; P: Pred a; Q: Pred b
7  . map f nil = nil
8  . map f (cons x l) = cons (f x) (map f l)
9  . all P nil
10 . all P (cons x l) <=> P x /\ all P l
11 . filter nil P = nil
12 . filter (cons x l) P = cons x (filter l P) when P x else filter l P
13 . map (\x:a. g (f x)) l = map g (map f l)          %(map_compose)%   %implied
14 . def map f l <=> all (\x:a. def f x) l            %(map_all_def)%   %implied
15 . def map f l => def (map f (filter l P))          %(mapdef)%        %implied
16 . def map f l =>
17   filter (map f l) Q = map f (filter l (\x:a. Q (f x))) %(mapfilter)% %implied
```

Listing 2.2: HasCASL basic specification for map-related operators on lists, demonstrating polymorphism, type constructors, and higher-order functions

- T is a set of type constructors, each associated with the set of the kinds it belongs to.
- A is a set of type synonyms, each associated with an expansion, i.e. a pseudotype formed like a λ-term with constants from T.
- O is a set of constants, each associated with a sort term.
- $\leq \subseteq T \times P$, where P is the set of pseudotypes over T, is the (explicit) subtype relation.

For an example of a HasCASL basic specification, see an adapted version of the Hets-lib[7] specification of some higher-order, polymorphic, map-related functions on lists in Listing 2.2. Lines 1 to 5 induce the below components of the signature Σ_{Map}. Note that C_{Map}, $\leq_{C_{Map}}$, A_{Map}, and O_{Map} contain built-ins induced by every HasCASL specification. The latter even contains infinitely many interpreted implicit function application operators (`__␣__`, basically the spaces between operation symbols) and partial upcast operators (`__ as [Supertype]`). Σ_{Map} should also make clear that the explicit subtype and subclass relations are only generators for the actual subkind and subpseudotype relations $\leq_K$ and $\leq_P$ respectively. These are each computed using a set of subkinding and subtyping rules and determine which classes or types may be used in places where other classes or types

[7] `https://github.com/spechub/Hets-lib`, specifically the file `HasCASL/Map.dol`

are expected.

$$
\begin{aligned}
C_{Map} &= \{\texttt{Type: Type}\} \\
\leq_{C_{Map}} &= \{(\texttt{Type}, \texttt{Type})\}, \\
T_{Map} &= \{\texttt{Unit: Type}, \\
&\quad \texttt{\_\_*\_\_: +Type -> +Type -> Type}, \\
&\quad \texttt{\_\_->\_\_: -Type -> +Type -> Type}, \\
&\quad \texttt{\_\_->?\_\_: -Type -> +Type -> Type}, \\
&\quad \texttt{List: Type -> Type}\}, \\
A_{Map} &= \{\texttt{Pred:=(}\lambda\texttt{a. a ->? Unit)}, \\
&\quad \texttt{Logical:=Pred Unit}\}, \\
O_{Map} &= \{\texttt{\_\_}\sqcup\texttt{\_\_: ((a -> b) * a) -> b} \mid \texttt{a, b} \in \texttt{Type}\} \\
&\cup \{\texttt{\_\_}\sqcup\texttt{\_\_: ((a ->? b) * a) ->? b} \mid \texttt{a, b} \in \texttt{Type}\} \\
&\cup \{\texttt{\_\_ as a: b ->? a} \mid \texttt{a, b} \in \texttt{Type} \wedge \texttt{a} \leq_P \texttt{b}\} \\
&\cup \{\texttt{(): Unit}, \\
&\quad \texttt{\_\_,\_\_: (}\forall\texttt{a, b: Type. a -> b -> a * b)}, \\
&\quad \texttt{nil: (}\forall\texttt{a: Type. List a)}, \\
&\quad \texttt{cons: (}\forall\texttt{a: Type. a -> List a -> List a)}, \\
&\quad \texttt{map: (}\forall\texttt{a, b: Type. (a ->? b) -> List a ->? List b)}, \\
&\quad \texttt{all: (}\forall\texttt{a: Type. Pred a -> List a -> Logical)}, \\
&\quad \texttt{filter: (}\forall\texttt{a: Type. List a -> Pred a -> List a)}\} \\
\leq_{Map} &= \{\}
\end{aligned}
$$

Lines 6 to 12 (with variables from line 2) of Listing 2.2 further provide a set of axioms Γ_{Map} defining the operations' behaviour using recursion. Finally, lines 13 to 18 (still borrowing variables form line 2) postulate some properties that are supposedly implied by the definitions of Γ_{Map}. These illustrate some of the ways for constructing sentences with HasCASL. In general, a $SubPCoClTyConsProdsHOL^{=}$ sentence is a definedness assertion of or an existential or strong equation between fully-qualified terms of the same sort, as in λ_P-calculus [48]. λ_P is a generalization of simply-typed λ-calculus, where λ-abstraction may denote partial functions and definedness `def`[8], existential equality `__=e=__` and strong equality `__=__` of terms are interdefinable [74] atomic predicates:

- `def t` $\equiv$ `t =e= t`
- `t =e= s` $\equiv$ `t = s` $\wedge$ `def t` $\wedge$ `def s`
- `t = s` $\equiv$ (`def t` $\Rightarrow$ `t =e= s`) $\wedge$ (`def s` $\Rightarrow$ `t =e= s`)

Predicates on a type `a` are regarded as partial functions into `Unit`, i.e. values of type `Pred a` (cf. A_{Map} above). Predicates on `Unit` encode truth values and have their own type synonym, `Logical`. Definedness on a value signifies a satisfied predicate or, in the case of `Logical`, logical truth. HasCASL allows coating $SubPCoClTyConsProdsHOL^{=}$ sentences

[8] called "existence" `E` by Moggi and Scott

in various layers of syntactic sugar to make them look more like typical higher-order logic (HOL) formulas:

elementhood operators `x in S` $\mapsto$ `def x as S`,

total λ-abstractions `\x:X.! t` $\mapsto$ `(\x:X. t) as X -> T`

iterated abstractions `\x1:X1 x2:X2 ... xn:Xn. t` $\mapsto$ `\x1:X1.! \x2:X2.! ... \xn:Xn. t` and `\. t` $\mapsto$ `\x:Unit. t`, where `x` is not free in `t`,

procedural lifting `?T` $\mapsto$ `Unit ->? T`, with the replacements `t: T` $\mapsto$ `\. t: ?T` and `t: ?T` $\mapsto$ `t(): T` implicitly being performed whenever a value of the respective type is expected but one of the other type was given[9],

let-terms `let x = s in t` $\mapsto$ `(\x. t)s` $\dashv$ `t where x = s`,

patterns in let-terms allow projecting the components out of values of a product type and deconstructing members of datatypes, e.g. `let (x,y) = s in t` where `s: X * Y`,

restriction operators `s res t` $\mapsto$ `let (x,y) = (s,t) in x`, which is defined iff both `s` and `t` are defined or, in the case of `t: Logical`, iff both `s` and `t()` are defined

From this, the usual logical operations are constructed:

logical truth `tt` $\mapsto$ `()`

conjunction `/\` $\mapsto$ `res`

implication `s => t` $\mapsto$ `((\. s) =e= \. (s /\ t))`

biconditional `s <=> t` $\mapsto$ `(s => t) /\ (t => s)`

universal quantification `forall x: X. t` $\mapsto$ `((\x:X. t) =e= \x:X. tt)`

logical falsehood `ff` $\mapsto$ `forall z: Logical. z()`

negation `not t` $\mapsto$ `t => ff`

disjunction `s \/ t` $\mapsto$ `forall z: Logical. ((s => z()) /\ (t => z())) => z()`

existential quantification
`exists x: X. t` $\mapsto$ `forall z: Logical. (forall x: X. t => z()) => z()`

A $SubPCoClTyConsProdsHOL^{=}$ model for a signature Σ is a derived signature morphism $\sigma\colon \Sigma \to \Sigma'$ paired with a λ_p-algebra that is admissible for Σ'. An admissible algebra for a signature Σ' satisfies a λ_p-theory $Th(\Sigma')$ induced by that signature. The details of this construction have been described by Schröder and Mossakowski [72, 52], but are not relevant for this work. One important consequence of this construction is that it permits an equivalent logical interpretation of HasCASL due to the equivalence of λ_p-algebras to intensional Henkin models [37, 32]. [70] This interpretation gives HasCASL an intensional intuitionistic logic without choice operators, although classicality, extensionality, unique choice, and Hilbert choice can be specified if desired. With this in mind, HasCASL formulas can be handled directly as the expected logical formulas without the need for a translation into $SubPCoClTyConsProdsHOL^{=}$ sentences.

[9] This applies (to) values of type `Logical` as well, since it is just `?Unit`.

2.3 Tons of Inductive Problems

The Tons of Inductive Problems (TIP) format in its current form is an extension of a subset of the SMT-LIB script language, version 2.6. [40] SMT-LIB is used to describe satisfiability modulo theories (SMT) problems, by defining: [3]

terms and formulas of many-sorted first-order logic with equality, with formulas being terms of the distinguished sort `Bool`,

theories that fix a vocabulary of sorts, functions, and predicates, like a `Bool` sort and some of the logical connective symbols that are part of the `Core` theory which is implicitly used by every SMT-LIB theory,

logics that fix one or more theories, may restrict the language of formulas in instances of the, possibly combined, theory, and may define syntactic sugar as extensions to that language, and

commands to communicate to an SMT solver what it should work on, including what logic to use and the formulas to check for satisfiability.

SMT-LIB scripts are sequences of commands. Commands are written in a LISP-like syntax, meaning that every command is a parenthesized list of command name and the corresponding number of arguments. The SMT-LIB script commands allowed in the TIP format are:

- `assert` to add a new formula that may restrict satisfiability,
- `declare-sort` for adding a new sort symbol of given arity,
- `declare-const`, `declare-fun` to add function symbols with given rank, i.e. combination of domain sorts and codomain sort,
- `define-fun` as a shorthand for `declare-fun`ing a function symbol and `assert`ing its values on all instances of its domain sorts, without using the newly-declared function symbol on the right-hand side of the definition,
- `define-fun-rec`, `define-funs-rec` to do the same for one or more (possibly mutually) recursive function definitions, and
- `declare-datatype`, `declare-datatypes` to introduce one or more (possibly parametric and possibly mutually recursive) algebraic datatypes by `declare-sort`ing their names and respective numbers of parameters, `declare-fun`ing their respective constructors and `define-fun`ing selectors, or at least planning these for instantiation in the parametric case.

Note that SMT-LIB only allows the declaration of parametric datatypes, but not their uninstantiated use. Since function declarations cannot make use of sort parameters with which to instantiate a parametric datatype, parametrically polymorphic functions cannot be expressed. It is, however, possible to `declare-fun` the same function symbol multiple times with differing ranks, unless it was already part of the underlying theory, thus enabling ad-hoc polymorphism.

Since many of the inductive problems in the TIP library also involve parametrically polymorphic functions and higher-order functions, Claessen et al. saw reason to include these as features in their format. [21] This was implemented by

- fixing the sort symbol `=>` for arbitrary arities to represent sorts of functions,
- fixing the function symbol `@` to apply members of function sorts to their arguments[10],
- adding an abstraction binder `lambda` to construct members of function sorts in a similar way to the definition part of `define-fun`,
- allowing sort parameters into assertions and function declarations, and
- making polymorphic function symbols indexable to explicitly instantiate them at given sorts[11].

Because theorem proving was not the main scope of SMT-LIB, Claessen et al. also added the `prove` command to explicitly mark proof (sub)goals. Theorem proving could be emulated in SMT-LIB scripts through use of the assertion stack and `assert`ing each potential theorem's negation on a new level of the stack. This would coerce the solver into finding counterexamples for the theorems' negations, thus producing witnesses for refutation proofs of the original theorems.

While the semantics of SMT-LIB scripts have been thoroughly defined by Barrett, Fontaine, and Tinelli, TIP format extensions lack such a definition. Higher-order functions are compatible with the SMT-LIB semantics, as `=>` and `@` can be fixed in an SMT-LIB theory and `lambda` can be seen as syntactic sugar[12] introduced in an SMT-LIB logic. Parametric polymorphism, however, does not fit into this framework and the transformation of a polymorphic first-order formula into a finite and equisatisfiable set of monomorphic first-order formulas is undecidable. [11]

2.4 Zipperposition

Zipperposition is an ATP built around Logtk, a toolkit for rapid prototyping of new ideas for proof calculi. Logtk itself offers a calculus based on superposition for polymorphic FOL. [23, 24] The first iteration of Zipperposition was an implementation of Cruanes's extensions of superposition for handling arithmetic and induction. [24, 25] It was later extended by calculi for λ-free clausal HOL [7], clausal HOL [8], FOL with first-class Booleans [58], and full HOL with first-class Booleans [9]. More precisely, the latter extension operates in a "[HOL] (simple type theory) with rank-1 polymorphism, Hilbert choice, and functional and Boolean extensionality", while the simple induction extension adds induction schemas for inductive types to Logtk's FOL.

[10] It follows that the application symbol must permit all ranks that start with a function sort and end with that function sort's rank.

[11] Type inference only considers the function symbol's associated ranks and the function application's input sorts. Explicit instantiation is therefore necessary where this yields multiple possible output sorts.

[12] Anonymous functions can be extracted into named function definitions.

CHAPTER 3

Related Work

In this chapter I will give a brief overview over existing work related to logic translations and interoperability between theorem provers.

3.1 Tools for Inductive Provers

The TIP library was not created with the expectation that its problem format (cf. Section 2.3) would be supported by every inductive prover in existence. Due to differences in the supported featuresets of different provers, this would have been impractical, too. The format was therefore explicitly designed as a union of the featuresets of inductive provers with translatability in mind. For this purpose, the Tools for Inductive Provers (TIP tools[1]) have been provided alongside the TIP library. [64] The TIP tools can translate[2] problems expressed in the TIP format to the Why3 [13] specification language, to SMT-LIB [3] scripts (cf. Section 2.3), to Isabelle [60] theories, to the Thousands of Problems for Theorem Provers (TPTP) [78] format in typed first-order form (TFF) [79], to Waldmeister's [18] input format, and to Haskell testing specifications for use with QuickCheck [20], Feat [29], LazySmallCheck [68], and Smten [82]. To accomodate the target formats that lack support for some TIP format features, encodings are used. Each of these encodings, as well as further transformations, can be applied independently of target format. Due to the undecidability of monomorphisation mentioned in Section 2.3, the polymorphism encoding step is limited by a configurable number of rounds and may fail.

Adding a TIP format backend to Hets will open up the possibility of using the TIP tools for further translations. This will hopefully lower the burden of connecting ATPs and model finders that Hets currently cannot talk to. In this work, I will not incorporate the TIP tools, instead testing just the TIP format backend using an ATP that understands the TIP format natively.

[1] Rosén and Smallbone used TIP as an abbreviation for both the library, the format, and the tools, which I will try to avoid by only abbreviating Tons of Inductive Problems.

[2] This was taken from the listing of available output formats of version 0.2.2 of the TIP tool `tip`.

3.2 Why3

The Why3 environment consists of a specification language, the programming language WhyML, and tools for extracting and managing proof obligations from specifications and programs. [14] The Why3 specification language is based on many-sorted first-order logic with parametric polymorphism, recursive function definitions, algebraic datatypes, and inductive predicates. Why3 specifications are grouped into theories which can be imported by other specifications and WhyML specification annotations. As a peculiarity[3], the built-in theory `HighOrd` introduces a function type constructor and a function application operator, thus smuggling higher-order features into the first-order syntax. WhyML incorporates specifications to annotate, for example, function implementations with pre- and postconditions, loops with invariants and recursive functions and `while`-loops with termination measures. The Why3 tools offer both a command-line and a graphical interface for batch and interactive processing of proofs, respectively. These tools can dispatch proof obligations to the ATPs Alt-Ergo [12], Beagle [6], CVC3 [4], CVC4 [5], E [73], Gappa [26], Metis [39], MetiTarski [1], Princess [67], Psyche [36], Simplify [27], SPASS [84], Vampire [63], veriT [16], Yices 1 [31] and 2 [30], and Z3 [56] and to the interactive provers Coq [22], PVS [59], and Isabelle/HOL [57]. Verified WhyML programs can also be transalated to correct-by-construction OCaml [62] programs.

As with the TIP tools, adding a Why3 backend to Hets would ease the integration of many as-yet unconnected theorem provers. Theorem provers are also already integrated into the Why3 environment, whereas the TIP tools only offer raw translation and no connection to provers of their own. It is, however, possible to translate specifications from TIP format to Why3 format, so that the implementation of a TIP backend seems like a logical first step toward harnessing the brokering abilities of Why3.

3.3 Sledgehammer

Sledgehammer is a component of the interactive theorem prover Isabelle/HOL [57] that uses an ensemble of ATPs as a relevance filter for facts to pass to its internal Metis implementation [61] for proof finding. [15] It heuristically pre-selects hundreds of facts, then tasks its connected provers with proving the current goal given these facts and finally extracts the facts that were actually used by successful proof attempts from these attempts' outputs. Version 2021-1 of Sledgehammer makes use of agsyHOL [45], Alt-Ergo [12], CVC4 [5], E [73], iProver [43], LEO-II [10], Leo-III [76], Satallax [17], SPASS [84], Vampire [63], veriT [16], Waldmeister [18], Z3 [56], and Zipperposition [23].

Unlike Hets and Why3, which use their connected provers to discharge proof obligations directly, a successful proof attempt by a Sledgehammer-connected prover is not sufficient for discharging a goal in Isabelle. A goal is only proven if Metis, given Sledgehammer's output, finds a proof that can be translated into a sound Isabelle proof. This affords Sledgehammer the freedom to use unsound translations, which may result in a lower workload for connected provers, but makes their provability judgments untrustworthy on their own.

Hets is already connected to Isabelle 2014 and can therefore make use of that version's

[3] This is not mentioned in the cited sources and had to be inferred from examples and source code.

Sledgehammer-supported proof search. Version 2014 of Sledgehammer lacks connections to Leo-III, veriT and Zipperposition. It has additional connections to E-SInE [38], E-ToFoF[4], iProver-Eq [44], SNARK [77], and CVC3 [4]. While the latter two seem to have been cut without replacement, iProver has since gained native equality handling [28] and E has since gained its own implementation of SInE axiom selection and native TFF support [73], making iProver-Eq, E-SInE, and E-ToFoF obsolete, respectively. Updating the Isabelle interface to support version 2021-1 would therefore improve Hets' ability to discharge higher-order proof obligations via the new ATPs within Sledgehammer, but only in cases where the incomplete higher-order calculus within Metis happens to find a proof as well. A direct connection of a higher-order ATP to Hets is therefore complementary to any improvements brought about by Sledgehammer updates.

[4] A script for converting TFF problems to first-order form (FOF) ones and running E on the result, according to the archived version of `http://www.cs.miami.edu/~tptp/ATPSystems/ToFoF/`.

CHAPTER 4

Thesis Contribution

The ubiquity of inductive datatypes in real-world applications has been established in Chapter 1 and emphasized by their inclusion in otherwise first-order specification languages like CASL and SMT-LIB. In the following, I will address the question whether Hets benefits from incorporating Zipperposition as an ATP for handling proof obligations involving such datatypes. To that end, I will develop TIP printers for CASL and HasCASL, which is a potential benefit in itself, as explained in Section 3.1. After connecting Zipperposition with such a printer, I will determine proof obligations can be discharged automatically which could not be before. But first, a small result has to be noted that is crucial when trying to find algebraic datatypes to represent in TIP specifications.

4.1 Strengthening partiality encoding

Both $(4')$ and $(4a')$ from Section 2.2.3 add nullary constructors $\bot$ to sort generation constraints, which they assert to be undefined. They do not assert that their images are disjoint from the existing constructors', and in the case of $(4a')$, this would even introduce a contradiction for any constructor with an argument of the result sort. But even $(4')$ does not preserve the axioms asserting injectivity and disjointness of images of constructurs that are introduced by free type declarations. Because they are universally quantified sentences, they are relativized to only hold on defined terms. If free types in a specification with partial functions should keep their interpretation as free types despite partiality encoding, injectivity and disjointness of constructors have to be preserved, even for $\bot$. The following takes place completely within $PCFOL^=$, so the last component of a signature is always the equality on the set of sorts, and will therefore be omitted for brevity.

For a flat $PCFOL^=$ specification (Σ, Γ) defining only freely and loosely interpreted sorts, this can be achieved by the following process: Let $(S_l \oplus S_f, TF_c \oplus TF_r, PF, P) = \Sigma$ be the components of the signature, where

- S_l is the set of loosely interpreted sorts,
- S_f is the set of freely interpreted sorts,
- TF_c is the set of constructors for the freely interpreted sorts, and
- TF_r is the set of remaining total operations.

Let furthermore $\Gamma_c \oplus \Gamma_i \oplus \Gamma_d \oplus \Gamma_r = \Gamma$ be the sort generation constraints, injectivity axioms, disjointness axioms, and remaining sentences, respectively, of the theory. Now let

- SP_l be the basic specification consisting only of declarations for the sorts in S_l,
- SP_f be the basic specification consisting only of free datatype declarations for the sorts of S_f with constructors from TF_c, but without any selectors,
- SP_F be the basic specification consisting only of loose datatype declarations for the sorts of S_f with the constructors from TF_c, but without any selectors, and
- SP_r be the basic specification consisting only of declarations for the elements of TF_r, PF, P and assertions of the sentences in Γ_r.

Then both the structured specifications

$$SP_l \texttt{ then } SP_f \texttt{ then } SP_r \tag{4.1}$$

$$SP_l \texttt{ then free \{ } SP_F \texttt{ \} then } SP_r \tag{4.2}$$

determine the same model class as (Σ, Γ). For spec. 4.1 this is obvious, because it is just a modularized version of (Σ, Γ), where the sentences in Γ_c, Γ_i, and Γ_d have been made obsolete by the free datatype declarations in SP_f. The equivalence of specs. 4.1 and 4.2 is a result from Section 2.2.1. The first extension in specification spec. 4.2 is a free extension free (SP_l `then` SP_F) along σ, where

$$\sigma\colon (S_l, \emptyset, \emptyset, \emptyset) \to (S_l \cup S_f, TF_c, \emptyset, \emptyset)$$

is the embedding of loose sorts into the extension by the free sorts and their constructors.

Now, $\gamma^{(4')}$ can be checked for σ-normality by considering $\mathrm{Mod}(\sigma)$ and $\mathrm{Mod}(\Phi^{(4')}(\sigma))$. Note that, technically, $CFOL^=$ is defined as the intersection of $SubCFOL^=$ and $PCFOL^=$, where the latter is defined via a reduction to the former. This is why the model functor is the same after partiality encoding. Models for $CFOL^=$ can still contain partial functions, but they are superfluous and such models are isomorphic to the ones leaving just them out. Firstly, $\mathrm{Mod}(\sigma)$ simply forgets the carriers of the sorts in S_f and erases all functions. $\gamma^{(4')}_{(S_l,\emptyset,\emptyset,\emptyset)}$ adds just the undefined value $\bot$ to each carrier, adds an interpretation for the $\bot_s$ constants as that sort's $\bot$, and interprets the definedness predicate by the original carriers. The composition $\gamma^{(4')}_{(S_l,\emptyset,\emptyset,\emptyset)} \circ \mathrm{Mod}(\sigma)$ therefore needs no explanation. Furthermore,

$$\begin{aligned}\Phi^{(4')}(\sigma)\colon (S_l, \{\bot_s\colon s \mid s \in S_l\}, \emptyset, \{D_s\colon s \mid s \in S_l\}) \\ \to (S_l \cup S_f, TF_c \cup \{\bot_s\colon s \mid s \in S_l \cup S_f\}, \emptyset, \{D_s\colon s \mid s \in S_l \cup S_f\})\end{aligned}$$

is the analogous embedding of loose sorts, their $\bot_s$ and definedness predicates into the extension by the free sorts. $\mathrm{Mod}(\Phi^{(4')}(\sigma))$ therefore forgets the carriers of the sorts in S_f, their subsets corresponding to defined individuals, and all the functions with result sorts in S_f. $\gamma^{(4')}_{(S_l \cup S_f, TF_c, \emptyset, \emptyset)}$ adds the undefined value $\bot$ to each carrier and adds constructor applications involving any $\bot_s$ to carriers for sorts in S_f, while again interpreting the definedness predicate with original carriers. Finally, $\mathrm{Mod}(\Phi^{(4')}(\sigma)) \circ \gamma^{(4')}_{(S_l \cup S_f, TF_c, \emptyset, \emptyset)}$ leaves only the carriers of sorts in S_l, augmented by the $\bot$, the interpretations of the $\bot_s$ as the only functions, and the original carriers for sorts in S_l as interpretations for the definedness predicates. This is the same as $\gamma^{(4')}_{(S_l,\emptyset,\emptyset,\emptyset)} \circ \mathrm{Mod}(\sigma)$.

It is now possible to apply (4′) to spec. 4.2, which leads to

$$SP'_l \texttt{ then free \{ } SP'_F \texttt{ \} then } SP'_r, \text{ where} \tag{4.3}$$

- SP'_l is SP_l augmented by the constants $\bot_s$, the predicates D_s, and the axioms $\forall x \in s.\, D_s(x) \implies x \neq \bot_s$ and $\exists x \in s.\, D_s(x)$ for each $s \in S_l$,
- SP'_F is SP_F augmented by the constants $\bot_s$, the predicates D_s, and the axioms $\forall x \in s.\, D_s(x) \implies x \neq \bot_s$ and $\exists x \in s.\, D_s(x)$ for each $s \in S_f$, as well as axioms for reflection and preservation of definedness for constructor applications, which may involve applications of the definedness predicates for sorts in S_l,
- SP'_r is a full application of (4′), including sentence translations like relativized quantifiers.

The definedness predicates and the axioms about it in SP'_F can be moved to a subsequent extension SP^2_F without changing their interpretation, because they do not restrict the carriers for the free types. This leaves the remaining specification SP^1_F with datatype declarations including the $\bot_s$ constructors:

$$SP'_l \texttt{ then free \{ } SP^1_F \texttt{ \} then } SP^2_F \texttt{ then } SP'_r \tag{4.4}$$

Because SP^1_F does not declare sorts from S_l or involve total selectors, the free specification over it can be replaced with SP^1_f where all datatype declarations have been replaced with free datatype declarations:

$$SP'_l \texttt{ then } SP^1_f \texttt{ then } SP^2_F \texttt{ then } SP'_r \tag{4.5}$$

Now, specs. 4.3 to 4.5 describe the same model class and spec. 4.5 is again just a modularized version of a flat specification (Σ', Γ') with the following properties:

- It is $CFOL^=$, since the steps since spec. 4.3 have not introduced any partial functions.
- The sorts in S_f are interpreted as free types; injectivity and disjointness for constructors holds even on undefined values, since these have been unpacked from the free datatype declarations after applying (4′).
- There are no proper selectors for the free types in S_f, since their axiomatization was relativized in SP'_r. Putting them into the datatype declarations in SP_f or SP_F would not have changed that since their axiomatization would need to be unpacked before applying (4′).

4.2 Prerequisites

Chapter 2 has shown that specification languages can be equipped with vastly different semantics and that provers implement calculi whose soundness depends on the properties of a certain semantics. For the sake of efficiency or interoperability, even pre-defined semantics may be adapted while retaining the syntactical features of a specification language, e.g. the variants of the HasCASL institution used by Schröder and Mossakowski to establish a context of compatibility with CASL. One can therefore not simply unleash any prover on every proof, even if it is formulated in a language that the prover “understands”, and expect

a proof that reflects the author's intent. I will therefore first examine how the TIP format could express sentences of the CASL and HasCASL institutions and then check under which circumstances Zipperposition can produce sound proofs for the original problems.

4.2.1 Literal translations

Since specifications in Hets are represented in terms of their institutions, it is sufficient to consider ways of expressing the syntactic components of institutions, namely their signatures and sentences.

CASL

The TIP format cannot express subsorting, so it can at most express specifications of $PCFOL^=$. Because the subinstitution comorphism $(3')$ from Section 2.2.2 is already implemented in Hets, borrowing of provers is not restricted to specifications without subsorting.

Another complication is the fact that the TIP format is supposed to treat all functions as partial, but does neither offer a definedness predicate nor strong or existential equality. Since Zipperposition's interpretation of the format treats functions as total[1] and the TIP tools' translation to SMT-LIB simply completes inexhaustive match constructs to make functions total, it is safest to avoid that "feature" and assume all functions to be total. The TIP format can thus only be used as an expression of the $CFOL^=$ subinstitution. The comorphisms $(4')$ and $(4a')$ from Section 2.2.3 are in a way already implemented in Hets. Indiscriminately applying $(4a')$ to every $PCFOL^=$ specification would demote free types to generated ones. As explained in the translation of sort generation constraints below, this would make them intractable for Zipperposition's first-order mode with induction. Fortunately, the restriction of $(4')$ that all constructors in sort generation constraints have to be total is weaker than the one FOL mode imposes on specifications, since free types cannot have partial constructors to begin with. Unfortunaly, the decision whether $(4')$ was applicable when it was requested as an encoding in Hets was not specific enough. It was not possible to apply $(4')$ to specifications that contained any sort generation constraints at all, not just ones that had partial constructors. I call the subinstitution of $SubPCFOL^=$ that does not permit partial constructs in sort generation constraints $SubPtCFOL^=$. Here the tC stands for a restriction of the sort generation constraint feature to total constructors. Introduction of this subinstitution to Hets is detailed in Section 4.3.1. For specifications that fit in $SubPtCFOL^=$, the user can thus choose between comorphism paths that apply either $(4')$ or $(4a')$ to get rid of partiality and end up with a specification in $CFOL^=$.

The components of a $CFOL^=$ signature $(S, TF, \emptyset, P)$ can be matched to TIP constructs as follows:

S: Sorts are introduced by the `declare-sort` and the `declare-datatype` commands. The former introduces them as loose types, which, in the absence of sort generation constraints, is the correct interpretation for elements of S. The latter can only be considered in connection with sentences, as discussed below.

[1] $\forall x\colon S.\, \exists y\colon T.\, f(x) = y$ is a theorem in a specification with sorts S, T, function symbol $f\colon S \to T$ and no sentences.

```
1  (declare-datatype Nat1 ((Zero1) (Suc1 (Pred1 Nat1))))
2  (declare-sort Nat2 0)
3  (declare-fun Zero2 () Nat2)
4  (declare-fun Suc2 (Nat2) Nat2)
5  (declare-fun Pred2 (Nat2) Nat2)
6  (assert
7    (forall ((P (=> Nat2 Bool)))
8      (=>
9        (and (@ P Zero2) (forall ((n Nat2)) (=> (@ P n) (@ P (Suc2 n)))))
10       (forall ((n Nat2)) (@ P n)))))
11 (assert
12   (forall ((a Nat2) (b Nat2))
13     (=> (= (Suc2 a) (Suc2 b)) (= a b))))
14 (assert
15   (forall ((a Nat2))
16     (not (= Zero2 (Suc2 a)))))
17 (assert
18   (forall ((a Nat2))
19     (= a (Pred2 (Suc2 a)))))
```

Listing 4.1: Two equivalent TIP translations of `free type Nat ::= Zero | Suc(Pred:? Nat)`

TF: Function symbols are introduced by the `declare-{const,fun}`, `define-fun{,-rec}`, and `declare-datatype` commands. The two definition commands will not be used, since $SubPCFOL^=$ handles function definitions as separate sentences. The last command will be used if an operation is found to be a constructor of a free type, but this can only be determined in connection with sentences, as discussed below.

P: Predicates are just functions with result type `Bool` and can thus be treated analogously.

$SubPCFOL^=$, and therefore also $CFOL^=$, sentences are represented in Hets by data structures modelling the abstract syntax of CASL formulas [55, Sec. 2.2.1]. Unlike in the abstract syntax, sort generation constraints are treated as formulas as well and have a flag indicating if they come from a free type declaration. Most translations are straightforward and need not be elaborated here, except for two cases. Unique existential quantification has no direct counterpart in the TIP format and has to be circumscribed every time it is encountered. Sort generation constraints are translated as second-order induction axioms[2] and used to identify free datatypes, as illustrated separately in Listing 4.1. For a non-freely generated type, `Nat1` would be too strong of a statement and `Nat2` would be missing the assertions in lines 11 to 19.

HasCASL

Once again, the TIP format cannot express subsorting and does not guarantee that partial functions are interpreted as such. Furthermore, there is no support for type constructor classes or interpreted product type constructors. Of the subinstitutions of $SubPCoClTyConsProdsHOL^=$ known to Hets, this leaves polymorphic higher-order logic with type constructor definitions and equality ($PolyTyConsHOL^=$) as the most powerful that the TIP format can fully express. Presumably because of its integral role in the HasCASL type system, the subinsti-

[2] Which makes them invisible to the FOL-with-induction mode of Zipperposition. For example, $\forall a : \mathbb{N}.\, a = 0 \vee \exists b : \mathbb{N}.\, a = suc(b)$ can be refuted for `Nat2` but not (in reasonable time) for `Nat1` in Listing 4.1.

tution implementation in Hets does not allow classifying a specification into a subinstitution without the product type constructor if it defines any type constructors at all. This could be fixed with a theoroidal comorphism that encodes each n-product type constructor as an explicit n-ary type constructor with corresponding formation and projection operators, i.e. a free polymorphic type with n type variables, an n-ary instance constructor as tuple formation and each instance selector as one of the n projections.

4.2.2 Semantic compatibility

After smoothing out syntactical differences between specification and input format for a prover, it is still not clear whether the prover argues according to the specifier's intention. A typical example of an unintended inference would be the baseless introduction of tertium non datur into a proof about a specification that was written with an intuitionistic logic in mind. In the following, I will argue about the compatibility of Zipperposition's reasoning with the two specification languages' semantics.

CASL

Mossakowski established that $SubPtCFOL^{=}$ can be encoded in $CFOL^{=}$ in such a way that it is possible to reuse first-order theorem provers with induction (or second- or higher-order) for entailments concerning structured specifications including certain `free {…}` constructs. [50, Theorem 4.13] As explained in Sections 2.2.2 and 2.2.3, $(4') \circ (3') \colon SubPtCFOL^{=} \to tCFOL^{=}$ is strongly persistently liberal, thus achieving such an encoding. Another result is that $SubPCFOL^{=}$ can be encoded in $CFOL^{=}$ to allow reuse of first-order theorem provers with induction (or second- or higher-order provers) for basic specifications. [50, Theorem 4.8] The comorphism for this encoding only needs to be model-expansive, thus $(4a') \circ (3') \colon SubPCFOL^{=} \to CFOL^{=}$ is sufficient. For specifications consisting of the constructs introduced in Section 2.2, reuse of Zipperposition's higher-order mode is therefore always possible. The only limitation lies in its first-order mode's disregard for second-order sentences and the lack of another way to express induction axioms for non-freely generated datatypes.

HasCASL

As explained in Section 2.2.4, models for the logical interpretation of HasCASL are not required to be extensional, classical or to satisfy any choice principle. Bentkamp et al. explicitly mention the reliance of Zipperposition's higher-order calculus on extensionality and Hilbert choice. They also mention that they consider an equation like $t \not\approx \bot$, i.e. formula t is not violated, equivalent to $t \approx \top$, i.e. formula t is satisfied, and $t \not\approx \top$ equivalent to $t \approx \bot$, thus implying an assumption of classicality. It is sound for Henkin semantics. [9] The higher-order mode of Zipperposition can therefore only be used to reason about specifications that refine the specifications of functional extensionality and Hilbert choice[3] mentioned by Schröder and Mossakowski [72], either explicitly through structured specification constructs or implicitly through their own theories. The explicit case relies on a

[3] Hilbert choice and extensionality each imply classicality.

standard repository of specifications that can be used to build structured specifications and which includes extensionality and Hilbert choice. Hets-lib has no specifications for either functional extensionality, classicality or Hilbert choice and so none of the included specifications can make use this explicit method for refining them. Since there is currently no ATP for $SubPCoClTyConsProdsHOL^{=}$, the implicit case cannot be checked as part of a test for a particular specification's compatibility with Zipperposition. As a last resort, the specification's theory could be searched for sentences that match the patterns of the specific sentences used by Schröder and Mossakowski. Together with the TIP format's limited expressiveness, this reduces the usefulness of Zipperposition as an ATP for HasCASL considerably. The main advantage over a connection to CASL would be higher-order functions and polymorphism. Due to the lack of a body of viable specifications for evaluation, the connection to HasCASL will not be part of this work.

4.3 Implementation

Due to the extensive framework provided by Hets, the implementation of the necessary changes to the partiality encoding and the integration of Zipperposition proved very easy. Only the translation of CASL specifications into the TIP format required some work, which is detailed in the following.

4.3.1 More specific partiality encoding

The stronger version of $(4')$ described in Section 4.1 has not yet been implemented. It would require a flag like the one for sort generation constraints that indicates that they come from a free datatype declaration for generic sentences. The result is, however, applied when forming datatype declarations for TIP specifications. This would otherwise be impossible after partiality encoding, even the strongly persistently liberal one, since the relativized injectivity and disjointness axioms would be too weak to enforce an interpretation as a free type. A small tweak to the implementation of $(4')$ that keeps the free-origin flag on sort generation constraints was necessary to allow this behaviour.

The problem with the existing implementation of partiality encoding was that it assumed that sort generation constraints always require uniqueness of the undefined element. As established in Section 2.2.3, this is only true if a constraint defines partial constructors. I therefore introduced another subinstitution flag to the static analysis for CASL specifications. This flag signifies whether a specification uses only free datatypes ("f" before "C"), generated datatypes with total constructors ("t" before "C"), or generated datatypes including partial constructors (the general case, just "C").

The introduction of fC was needed to define an appropriate subinstitution for the first-order mode of Zipperposition. Since free types can only have total constructors, this further distinction does not harm the rejection decision of the partiality encoding. It was however necessary to define existing comorphisms' interaction with the new flag. The comorphism $(4a')$, for example, cannot preserve free types, but always codes out partial constructors. It is therefore clear that this variant always produces tC specifications if sort generation constraints were present in the input specifications. The comorphism $(4')$, on the other hand, preserves free types, as explained above, and will thus preserve the classification of

sort generation constraints of its input specifications, since it is not applicable to those with partial constraints and does not introduce constraints to those without any. Other comorphisms that had to be updated were encodings of CSMOF and QVT [19] into CASL, since they produce sort generation constraints. The fact that they only ever introduce free types emphasizes the relevance of the fC distinction for real-world use-cases.

4.3.2 Representing CASL specifications in TIP format

Hets represents CASL specifications as tuples containing a signature representation next to a list of named formula representations. Signatures contain maps of operation and predicate names to sets of their associated profiles and flags indicating their totality where applicable (i.e. representations of TF, PF and P), as well as a map of sorts to their sets of supersorts (i.e. a representation of both S and $\leqslant$). As mentioned in Section 4.2.1, formulas are representations of abstract syntax trees for formulas and sort generation constraints as defined in *The Complete Documentation of the Common Algebraic Specification Language*. Additionally, there are structures for second-order universally quantified formulas to facilitate the expression of sort generation constraints as induction axioms. Lastly, there are provisions for temporary forms of sentences as encountered during parsing, which need not be considered when working with fully parsed specifications.

The maintainers of TIP provide a grammar in labelled Backus-Naur form. This can be used to generate various implementations of an abstract syntax, a parser, and a pretty printer for the format using BNFC [33]. For this work the generated abstract syntax and pretty printer implementation for Haskell were essential for constructing syntax trees for TIP specifications and making them readable for Zipperposition. Section 4.3.3 provides further detail on the compromises that had to be implemented to accomodate Zipperposition.

The translation of a signature consists mainly of the declaration of sort and function symbols. To avoid name clashes, every sort, operation, and predicate symbol gets a new prefix that distinguishes it from pre-defined types of the TIP format and from other operations and predicates with the same name but differing profile. Sorts get the prefix `s_`, operations get `f<profile>_`, and predicates get `p<profile>_`. Here, `<profile>` is the list of argument sorts delimited by asterisks (`*`) which, in the case of operations, is followed by an arrow (`->`) and the result sort. Empty argument sort lists result in an empty `<profile>` in the case of predicates and an ommission of the arrow for operations.

Because Zipperposition warned about shadowing of identifiers, it seemed necessary to avoid repeating the declaration of a sort as a datatype, and so at first only loose and non-freely generated sorts are declared. To make this distinction, the information provided by the signature is not sufficient and the free datatypes have to be filtered from the sentences declaring them as such. This is done by examining the sort generation constraints, which carry with them a flag indicating if they come from a free datatype declaration. Because the results from Section 4.1 have not been completely implemented, this is the only realiable way to identify free types, as existing injectivity and disjointness sentences may be too weak. At the same time, constructors are extracted[4] to complete the datatype declarations and to avoid re-declaring them together with the other operations. Extracting selectors, however,

[4]Remember from Section 2.2 that sort generation constraints always have to carry their generating operations.

turned out to be harder than expected, since there is no implementation of a comparison of formulas modulo variable names implemented. For the time being, selectors are always freshly generated by prepending to the generated constructor name an `i` (**i**nverse), followed by the index of the constructor argument it is projecting and an underscore. The original selectors are preserved with the rest of the non-constructor operations and their relationship to their constructors is still specified in the theory. Because datatypes are specified with sentences using already declared sorts in $SubPCFOL^=$, mutual recursion does not pose any problem on the institution level. When using language constructs to declare datatypes, however, it has to be taken into account, which is why both CASL and the TIP format allow non-linear visibility of sort declarations within the same declaration construct. For the translation, this becomes important when the free types and their constructors and selectors have to be written down. All free type declarations are thus always put in a single `declare-datatypes` command. Finally, the remaining operations and predicates are declared with names generated as explained above.

Translation of formulas consists mainly of walking down the abstract syntax tree and switching CASL constructs for those generated from the TIP grammar. Variables introduced by quantification are treated like nullary operators above, only with a prefix beginning with a capital `F`. Since there are apparently a few logic languages without unique existential quantification, a transformation function was already implemented that applied the following definition:

$$\begin{aligned} &\exists! x : X, \ldots, z : Z.\, P(x, \ldots, z) \\ \equiv\ &\exists x : X, \ldots, z : Z.\, P(x, \ldots, z) \wedge (\forall x' : X, \ldots, z' : Z.\, P(x', \ldots z') \Rightarrow x' = x \wedge \ldots \wedge z' = z) \end{aligned}$$

Sort generation constraints are first translated into second-order sentences by re-using an existing implementation for generating first-order instances of induction axiom schemata. Because the predicate in these instances could not be reliably extracted to universally quantify after the fact, the implementation was changed to use the exact generated predicate symbol for a surrounding second-order quantification. This leaves the original functionality as a special case that unwraps the second-order quantification again. The translation of second-order sentences introduces the need to keep track of quantified predicates because their application differs from the usual first-order predicates. For this purpose, the walk down the tree carries a set of quantified variables that grows with every second-order quantification and is queried every time a formula consists of a predicate application. If the corresponding predicate symbol is in the set, the special `@` operator is used for the application, otherwise the predicate is applied as usual.

Features beyond such second-order formulas and $CFOL^=$ throw an error during the translation. These include existential equations and the interpreted definedness predicates. Implicit injections of elements of subsorts in places where the supersort would be expected are not detected but should be rejected by the prover's typechecker.

Since Hets is able to extract the axioms that were actually used in a proof from a prover's output, they are annotated either with auto-generated identifiers or names given in the specification (cf. Listing 2.1). The TIP format allows keyword annotations with an optional value and the authors used the `axiom` keyword for naming axioms. The original names are thus stored as values to `axiom` annotations. The format allows for the use of otherwise interpreted characters through escaping, which was not part of the code generated by BNFC

and had to be implemented, as well as the decoding of such escape sequences.

4.3.3 Integrating Zipperposition

Zipperposition was chosen for this work because of its induction capabilities and its native support for the TIP format. However, the parser implementation was based on an older version of the format and a quick fix was provided by the maintainers of the GitHub repository.[5] Some format changes were overlooked, though, and so the integration in Hets has to accomodate missing features through prover-specific workarounds. Due to a modular approach, these workarounds do not affect other potential consumers of the TIP format backend and can even be reused and extended for provers that have quirks of their own. The following workarounds have been implemented for Zipperposition:

Removal of annotations Because the parser does not expect any tokens between a command and its arguments, annotations like axiom names cause a parse failure. These are therefore removed when preparing the problem file passed to Zipperposition. Zipperposition numbers the commands in a problem file, starting with 0, and identifies each axiom by the index of the command which introduced it. This is exploited to reconstruct the original names of axioms from proofs.

Splitting of simultaneous datatype declarations Parsing for the `declare-datatypes` command was not adapted during the fix mentioned above and so, at the time of writing, only the singular command can be used. Datatypes are therefore declared singularly in the order in which their sort generation constraints appeared. This makes it impossible to use mutually recursive datatypes and breaks datatypes whose sort generation constraints did not occur dependency-ordered in the original specification. The latter problem could be remedied with a dependency analysis during application of the workaround. Since that would only be a partial fix and would be made redundant by a proper overhaul of Zipperposition's TIP parser anyway, no such workaround has been implemented.

Zipperposition's ability to output its proofs in TPTP typed higher-order form (THF) is leveraged to extract proof graphs and the list of used axioms from a proof. TPTP parsing was already implemented in Hets and needed only a little extension to deal with Zipperposition's way of expressing an axiom's origin. However, predicate symbols containing non-alphanumeric symbols are not properly quoted in the THF output.[6] This breaks parsing for proofs that use predicate symbols originally containing non-alphanumeric characters and, due to the asterisk used as a delimiter for argument sorts, predicates with arity greater than 1. This can only be fixed on Zipperposition's side and currently leads to no proof graph being built and all axioms being considered vital for such proofs.

4.4 Experiments

To test the efficacy of Zipperposition as an ATP for discharging inductive proof obligations, CASL specifications were needed that contained proof obligations which FOL provers like E

[5] `https://github.com/sneeuwballen/zipperposition/issues/88`
[6] `https://github.com/sneeuwballen/zipperposition/issues/93`

[73] and SPASS [84] were not already able to handle. For this purpose, the specification seen in Listing A.1 was created with the goal of providing proof obligations that could not be solved by simply applying definitions and being naturally $fCFOL^{=}$. It was then extended to $PfCFOL^{=}$, $SubfCFOL^{=}$, and $SubPfCFOL^{=}$ to evaluate the effect of partiality encoding, subsorting encoding, and both at once on the same goals' provability. The resulting specifications can be seen in Listings A.2 to A.4, respectively. The proof obligations were tried to be discharged using E, SPASS, and Zipperposition in FOL mode with induction, Zipperposition in HOL mode with $(4a')$ for partiality encoding, and Zipperposition in HOL mode with $(4')$ for partiality encoding, each given a 60 s timeout. The Hets feature for passing on proven theorems as axioms in subsequent proof attempts was turned off to prevent escalating advantages in case one prover was able to prove one theorem that another could not. No repetitions are made because the resulting times are only informative and do not require statistical significance.

CHAPTER 5

Thesis Outcome

These are the conclusions that I have drawn from my work on integrating Zipperposition into Hets using TIP as an interchange format. I will first detail my experience with using TIP to express inductive CASL problems. Then I will present the results of my experiments with the specifications in Listings A.1 to A.4 and try to interpret them.

5.1 TIP for inductive CASL problems

Any application of an induction principle in a proof requires a justification in the underlying logical system's model theory. The TIP format, presumably inheriting parts of its model theory from SMT-LIB, provides this justification in the form of algebraic datatypes. These are any sorts with a non-empty set of constructors and each constructor is required to have a number of distinct selectors attached to it that is equal to its arity. The interpretation of an algebraic datatype and its constructors is that of an absolutely free structure generated by its surrounding non-algebraic sorts, for which every selector is an inverse of the constructor it is attached to, in its assigned input argument. [3] The requirement of invertible, i.e. injective, constructors is integral to the SMT-LIB interpretation of algebraic datatypes. The TIP format additionally allows the explicit formulation of second-order induction axioms.

CASL, on the other hand, can justify induction with its sort generation constraints. These restrict interpretations of included sorts and constructors similarly to SMT-LIB's absolutely free structures, but do not require every constructor to be invertible in every argument, or even be total. This allows the succinct specification of set-like datatypes whose instances can be equal even if constructed in distinct ways.

A prover that merely extends FOL by some form of induction principle is not necessarily equipped to interpret higher-order axioms and recognize them as justifications for the application of such a principle. It would therefore be a specification language's responsibility to facilitate making this information explicit, especially if its raison d'être is the expression of inductive problems. In the tradition of deviating from SMT-LIB, a little format change could enable the explicit declaration of generated types: Allowing constructors without selectors in datatype declarations would allow an interpretation where such constructors are not required to be injective and their images not disjoint from those of other constructors.

Another difficulty in using the TIP format is the lack of a semantics that could disambiguate

the meaning of language constructs. As it stands, a prover's interpretation of the format has to first be reverse-engineered before it can be used with TIP specifications. This is especially problematic when what little information there is about the interpretation is still ambiguous. A statement such as "TIP allows partial functions. SMT-LIB does not." [40] can mean that functions cannot in general be relied upon to have a value for every input or, as was probably the intention of the authors, that functions can be defined with non-exhaustive match constructs while still being total and simply underspecified. That only becomes apparent in the TIP tools' implementation of the translation to SMT-LIB, which does not fundamentally change functions or the way in which they are used. By default, no partiality encoding is done at all and there are two additional SMT-LIB translation modes that replace match constructs with their definitions [3, Expr. 3.5] or translate `define-fun` statements into declarations and axioms. The former leads to nested conditional expressions for which the last else-branch contains possibly undefined applications of selectors and the latter just moves match constructs around. Only with a separate translation flag are incomplete match constructs completed with a default case that yields some fresh, and therefore underspecified, constant of the result sort. Therefore, the default SMT-LIB translation can produce invalid scripts and both ways of treating incomplete match constructs lead to the interpretation as underspecified total functions. Contrary to the aforementioned statement, this is exactly the interpretation that SMT-LIB assigns selectors, which are inherently partial. Having such information available in a reference document could have saved some effort trying to devise a comorphism that would encode the distinction between total and partial functions into a setting with only partial functions.

5.2 Zipperposition for inductive CASL problems

Neither E nor SPASS were able to discharge any of the goals from Listings A.1 to A.4. That means that Tables 5.1 and 5.2 represent a definitive improvement over the status quo insofar as Hets can find theorems automatically now that could not have been found before. This has to be qualified, though, by the fact that the specifications with the highest success rates were tailored to the prover's featureset. Any kind of encoding of CASL features not natively supported by Zipperposition or the TIP format leads to a sharp decline in successful proof

Goal \ feature level	$fCFOL^{=}$	$PfCFOL^{=}$	$SubfCFOL^{=}$	$SubPfCFOL^{=}$
add_0_right	5 ms	103 ms	643 ms	567 ms
add_assoc	82 ms	–	–	–
add_suc_right	1072 ms	–	–	–
add_comm	3128 ms	–	–	–
flip_flip	–	–	–	–
balanced_existence	–	–	–	–
concat_assoc	188 ms	–	–	–
concat_nil_right	44 ms	151 ms	–	–
reverse_concat	–	–	–	–
length_concat	121 ms	–	–	–

Table 5.1: Zipperposition FOL-with-induction mode performance (– for timeout after 60 s)

Goal \ feature level	$fCFOL^{=}$	$PfCFOL^{=}$	$SubfCFOL^{=}$	$SubPfCFOL^{=}$
add_0_right	266 ms	501 ms	1577 ms	1943 ms
add_assoc	1756 ms	–	–	–
add_suc_right	251 ms	–	–	–
add_comm	–	–	–	–
flip_flip	–	–	–	–
balanced_existence	–	–	–	–
concat_assoc	3563 ms	–	–	–
concat_nil_right	197 ms	–	35 466 ms	–
reverse_concat	–	–	–	–
length_concat	–	–	–	–

Table 5.2: Zipperposition HOL mode (non-persistently-liberal partiality encoding) performance (– for timeout after 60 s)

attempts. Partiality encoding by itself made any goal infeasible that involved more than one universally quantified variable. This may have to do with the addition of a conjunction as a premise when there is more than one variable for which definedness can be asserted (cf. lines 29 to 32 in Listing A.6). The conjuncts in the premise may be harder to delineate when generating induction lemmas than a single predicate.

Since the encoding of subsorting introduces partial projection functions (cf. lines 12 and 25 in Listing A.9) which have to be encoded as well, HOL mode's timeout for concat_nil_right in the merely partial setting stood out and the goal was retried with a more generous time limit. After 264 s, HOL mode was able to find the proof for that goal, showing that it may be worth waiting beyond my one-minute timelimit.[1] As expected, the combination of subsorting and explicit partial functions does not have a great impact on performance for the one goal that was solvable in every setting. Because of this, another attempt to have HOL mode find a proof for concat_nil_right in the subsorted, partial setting was started. It timed out after 10 min and having already taken up about two thirds of the 16 GB of available main memory. This may be due to the additional selector functions increasing the number of potential induction lemmas to instantiate the induction axiom with. Hets does not allow the removal of signature components so that an attempt without the likewise unnecessary comparison predicates and operations was not undertaken.

The fact that there is only one undefined element per sort to consider also works in HOL mode's favour. With the persistently liberal partiality encoding, it was unable to solve any of the proof goals in time. I speculate that HOL mode does not consider all the information from datatype declarations like injectivity and disjointness of constructors and thus has a much harder time arguing about "successors" of undefined elements. It is, for example, "trivial" for FOL-with-induction mode to declare injectivity for `Suc1` from Listing 4.1, but HOL mode cannot seem to find a justification.

[1] The time has not been recorded in the table since it was achieved outside the set experiment parameters.

CHAPTER 6

Conclusion

This part compresses the knowledge gained over the course of this thesis. I will summarize my contributions and point out potential for further research that was outside the scope of this work. Progress on mainlining my implementation can be tracked on GitHub.[1]

6.1 Summary

For this thesis, I have integrated an ATP capable of inductive reasoning into the proof management software Hets. I have evaluated the TIP format as an interchange format to talk to more than just the one prover chosen for this work, which was Zipperposition. The TIP format is capable of expressing the entirety of $CFOL^{=}$ due to its higher-order features. It did, however, prove to be unable to express the fact that an induction schema can be applied to a datatype without resorting to second-order sentences, i.e. induction axioms. The availability of an induction schema for a datatype can be expressed succinctly if that datatype is supposed to be interpreted freely, i.e. as an absolutely free structure containing invertible constructors. Another problem is the lack of a well-defined semantics, thus leaving each prover room for interpretation of the language constructs, which have to be considered when connecting a prover.

While researching comorphisms that would encode subsorting and partiality features of CASL to fit into $CFOL^{=}$, I discovered that Mossakowski's persistently liberal comorphism for encoding partiality can be strengthened under certain circumstances: First-order sentences that distinguish a freely generated from a non-freely generated datatype do not need to be relativized to apply only to defined values. This improvement has not yet been implemented in the partiality encoding within Hets, but is already exploited when constructing datatype declarations during preparation of TIP input for Zipperposition.

Zipperposition was tested on problems that ATPs with existing Hets integrations were not able to solve. Besides an extension of a first-order calculus with inductive reasoning, Zipperposition also features a higher-order mode. This allowed testing of non-persistently-liberal encodings, which would produce the aforementioned non-freely generated datatypes. Both modes were quite successful in finding new theorems when no encoding was necessary. However, they struggled with partiality encoding, which complicates proof goals and bloats

[1] `https://github.com/spechub/Hets/issues/1502`

up theories, solving only the easiest of goals. Zipperposition's HOL mode was not even able to find these with the persistently liberal partiality encoding. Each of the modes was able to find proofs that the other one could not, which shows that it was useful to include both modes as options for the user to choose from. I expect them to be of help with the verification of specifications written in languages with direct comorphisms to $CFOL^{=}$, like CL. Unfortunately, the examples of CL specifications in Hets-lib either contained no proof goals involving sequence markers or could not be opened using Hets, so this was not tested. Specifications involving subsorting or partiality features will likely not benefit much. At least now, users have the option of trying to have an ATP discharge their inductive proof obligations.

Use of Zipperposition is hampered, though, by its patchwork TIP parser and incomplete TPTP proof output. It is not possible to assign names to axioms, which Zipperposition should use when outputting proofs. It is not possible to declare mutually-recursive datatypes. Datatypes which depend on other datatypes are also broken in cases where their sort generation constraints do not perchance appear in dependency order within the original CASL specification. Predicates with names containing non-alphanumeric symbols are not quoted, and thus prevent any attempt at parsing the proofs.

6.2 Future Work

Due to time constraints, this work does not include a case study for the effectiveness of the Zipperposition integration within a real-life scenario. Such a case study could explore whether any useful proofs can now be found automatically. A first candidate would be the COLORE graph theories, for which Mossakowski et al. needed to use Isabelle when evaluating their CL integration [54]. Another candidate would be the ATM example used recently by Rosenberger, Knapp, and Roggenbach, who had actually hoped for automation support when doing their UML integration [65]. In a cursory test with their specifications, I encountered problems with interdependent datatype declarations did not look into it further. This has to be fixed within Zipperposition itself, and the workaround within Hets disabled.

Furthermore, more inductive provers can be connected via the TIP format backend and even more via translations offered by the TIP tools, which are also available for intergration in Hets as a Haskell library. One of these translations can provide a connection to the Why3 environment, thus gaining the support of many provers – automatic and interactive – at once. All of these connections have to be carefully checked for faithfulness to the original intention, though, since the TIP format's semantics are not well-defined.

Lastly, the stronger partiality encoding introduced in Section 4.1 needs to be implemented in Hets. This was also not done due to time constraints.

Bibliography

[1] Behzad Akbarpour and Lawrence C. Paulson. "MetiTarski: An Automatic Prover for the Elementary Functions". In: *Intelligent Computer Mathematics*. Ed. by Serge Autexier et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 217–231. DOI: 10.1007/978-3-540-85110-3_18.

[2] Egidio Astesiano et al. "CASL: the Common Algebraic Specification Language". In: *Theoretical Computer Science* 286.2 (2002). Current trends in Algebraic Development Techniques, pp. 153–196. DOI: 10.1016/S0304-3975(01)00368-1.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard*. Tech. rep. Version 2.6. Department of Computer Science, The University of Iowa, 2017. URL: https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2021-05-12.pdf.

[4] Clark Barrett and Cesare Tinelli. "CVC3". In: *Computer Aided Verification*. Ed. by Werner Damm and Holger Hermanns. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 298–302. DOI: 10.1007/978-3-540-73368-3_34.

[5] Clark Barrett et al. "CVC4". In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 171–177. DOI: 10.1007/978-3-642-22110-1_14.

[6] Peter Baumgartner, Joshua Bax, and Uwe Waldmann. "Beagle – A Hierarchic Superposition Theorem Prover". In: *Automated Deduction - CADE-25*. Ed. by Amy P. Felty and Aart Middeldorp. Cham: Springer International Publishing, 2015, pp. 367–377. DOI: 10.1007/978-3-319-21401-6_25.

[7] Alexander Bentkamp et al. "Superposition for Lambda-Free Higher-Order Logic". In: *Automated Reasoning*. Ed. by Didier Galmiche, Stephan Schulz, and Roberto Sebastiani. Cham: Springer International Publishing, 2018, pp. 28–46. DOI: 10.1007/978-3-319-94205-6_3.

[8] Alexander Bentkamp et al. "Superposition with Lambdas". In: *Automated Deduction – CADE 27*. Ed. by Pascal Fontaine. Cham: Springer International Publishing, 2019, pp. 55–73. DOI: 10.1007/978-3-030-29436-6_4.

[9] Alexander Bentkamp et al. "Superposition for Full Higher-order Logic". In: *Automated Deduction – CADE 28*. Ed. by André Platzer and Geoff Sutcliffe. Cham: Springer International Publishing, 2021, pp. 396–412. DOI: 10.1007/978-3-030-79876-5_23.

[10] Christoph Benzmüller et al. “LEO-II - A Cooperative Automatic Theorem Prover for Classical Higher-Order Logic (System Description)”. In: *Automated Reasoning*. Ed. by Alessandro Armando, Peter Baumgartner, and Gilles Dowek. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 162–170. DOI: 10.1007/978-3-540-71070-7_14.

[11] François Bobot and Andrei Paskevich. “Expressing Polymorphic Types in a Many-Sorted Language”. In: *Frontiers of Combining Systems*. Ed. by Cesare Tinelli and Viorica Sofronie-Stokkermans. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 87–102. DOI: 10.1007/978-3-642-24364-6_7.

[12] François Bobot et al. “Implementing Polymorphism in SMT Solvers”. In: *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*. SMT ’08/BPR ’08. Princeton, New Jersey, USA: Association for Computing Machinery, 2008, pp. 1–5. DOI: 10.1145/1512464.1512466.

[13] François Bobot et al. “Why3: Shepherd Your Herd of Provers”. In: *Boogie 2011: First International Workshop on Intermediate Verification Languages*. Wrocław, Poland, 2011, pp. 53–64. URL: https://hal.inria.fr/hal-00790310.

[14] François Bobot et al. “Let’s verify this with Why3”. In: *Int. J. Softw. Tools Technol. Transf.* 17.6 (2015), pp. 709–727. DOI: 10.1007/s10009-014-0314-5.

[15] Sascha Böhme and Tobias Nipkow. “Sledgehammer: Judgement Day”. In: *Automated Reasoning*. Ed. by Jürgen Giesl and Reiner Hähnle. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 107–121. DOI: 10.1007/978-3-642-14203-1_9.

[16] Thomas Bouton et al. “veriT: An Open, Trustable and Efficient SMT-Solver”. In: *Automated Deduction – CADE-22*. Ed. by Renate A. Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 151–156. DOI: 10.1007/978-3-642-02959-2_12.

[17] Chad E. Brown. “Reducing Higher-Order Theorem Proving to a Sequence of SAT Problems”. In: *Automated Deduction – CADE-23*. Ed. by Nikolaj Bjørner and Viorica Sofronie-Stokkermans. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 147–161. DOI: 10.1007/978-3-642-22438-6_13.

[18] Arnim Buch, Thomas Hillenbrand, and Roland Fettig. “WALDMEISTER: High performance equation theorem proving”. In: *Design and Implementation of Symbolic Computation Systems*. Ed. by Jacques Calmet and Carla Limongelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 63–64. DOI: 10.1007/3-540-61697-7_6.

[19] Daniel Calegari, Till Mossakowski, and Nora Szasz. “Heterogeneous verification in the context of model driven engineering”. In: *Science of Computer Programming* 126 (2016). Selected Papers from the 17th Brazilian Symposium on Formal Methods (SBMF 2014), pp. 3–30. DOI: 10.1016/j.scico.2016.02.003.

[20] Koen Claessen and John Hughes. “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs”. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP ’00. New York, NY, USA: Association for Computing Machinery, 2000, pp. 268–279. DOI: 10.1145/351240.351266.

[21] Koen Claessen et al. "TIP: Tons of Inductive Problems". In: *Intelligent Computer Mathematics*. Ed. by Manfred Kerber et al. Cham: Springer International Publishing, 2015, pp. 333–337. DOI: 10.1007/978-3-319-20615-8_23.

[22] Thierry Coquand and Gérard Huet. "The calculus of constructions". In: *Information and Computation* 76.2 (1988), pp. 95–120. DOI: 10.1016/0890-5401(88)90005-3.

[23] Simon Cruanes. "Logtk: A Logic ToolKit for Automated Reasoning and its Implementation". In: *4th Workshop on Practical Aspects of Automated Reasoning, PAAR@IJCAR 2014, Vienna, Austria, 2014*. Ed. by Stephan Schulz, Leonardo de Moura, and Boris Konev. Vol. 31. EPiC Series in Computing. EasyChair, 2014, pp. 39–49. DOI: 10.29007/4z1m.

[24] Simon Cruanes. "Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond". PhD thesis. École polytechnique, Sept. 2015. URL: https://hal.archives-ouvertes.fr/tel-01223502.

[25] Simon Cruanes. "Superposition with Structural Induction". In: *Frontiers of Combining Systems*. Ed. by Clare Dixon and Marcelo Finger. Cham: Springer International Publishing, 2017, pp. 172–188. DOI: 10.1007/978-3-319-66167-4_10.

[26] Marc Daumas and Guillaume Melquiond. "Generating formally certified bounds on values and round-off errors". In: *Proceedings of the 6th Conference on Real Numbers and Computers*. Ed. by Vasco Brattka, Christiane Frougny, and Norbert Müller. Schloß Dagstuhl, Germany, pp. 55–70. URL: https://hal.inria.fr/inria-00070739.

[27] David Detlefs, Greg Nelson, and James B. Saxe. "Simplify: A Theorem Prover for Program Checking". In: *J. ACM* 52.3 (May 2005), pp. 365–473. DOI: 10.1145/1066100.1066102.

[28] André Duarte and Konstantin Korovin. "Implementing Superposition in iProver (System Description)". In: *Automated Reasoning*. Ed. by Nicolas Peltier and Viorica Sofronie-Stokkermans. Cham: Springer International Publishing, 2020, pp. 388–397. DOI: 10.1007/978-3-030-51054-1_24.

[29] Jonas Duregård, Patrik Jansson, and Meng Wang. "Feat: Functional Enumeration of Algebraic Types". In: *Proceedings of the 2012 Haskell Symposium*. New York, NY, USA: Association for Computing Machinery, 2012, pp. 61–72. DOI: 10.1145/2364506.2364515.

[30] Bruno Dutertre. "Yices 2.2". In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. Cham: Springer International Publishing, 2014, pp. 737–744. DOI: 10.1007/978-3-319-08867-9_49.

[31] Bruno Dutertre and Leonardo de Moura. "A Fast Linear-Arithmetic Solver for DPLL(T)". In: *Computer Aided Verification*. Ed. by Thomas Ball and Robert B. Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 81–94. DOI: 10.1007/11817963_11.

[32] Melvin Fitting. "Classical Logic—Semantics". In: *Types, Tableaus, and Gödel's God*. Dordrecht: Springer Netherlands, 2002, pp. 11–32. DOI: 10.1007/978-94-010-0411-4_2.

[33] Markus Forsberg and Aarne Ranta. “BNF Converter”. In: *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*. Haskell ’04. Snowbird, Utah, USA: Association for Computing Machinery, 2004, pp. 94–95. DOI: 10.1145/1017472.1017475.

[34] Kurt Gödel. „Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I“. In: *Monatshefte für Mathematik und Physik* 38.1 (Dez. 1931), S. 173–198. DOI: 10.1007/BF01700692.

[35] Joseph A. Goguen and Rod M. Burstall. “Introducing Institutions”. In: *Logics of Programs, Workshop, Carnegie Mellon University, Pittsburgh, PA, USA, June 6-8, 1983, Proceedings*. Ed. by Edmund M. Clarke and Dexter Kozen. Vol. 164. Lecture Notes in Computer Science. Springer, 1983, pp. 221–256. DOI: 10.1007/3-540-12896-4_366.

[36] Stéphane Graham-Lengrand. “Psyche: A Proof-Search Engine Based on Sequent Calculus with an LCF-Style Architecture”. In: *Automated Reasoning with Analytic Tableaux and Related Methods*. Ed. by Didier Galmiche and Dominique Larchey-Wendling. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 149–156. DOI: 10.1007/978-3-642-40537-2_14.

[37] Leon Henkin. “Completeness in the Theory of Types”. In: *The Journal of Symbolic Logic* 15.2 (1950), pp. 81–91. DOI: 10.2307/2266967. (Visited on 09/07/2022).

[38] Kryštof Hoder and Andrei Voronkov. “Sine Qua Non for Large Theory Reasoning”. In: *Automated Deduction – CADE-23*. Ed. by Nikolaj Bjørner and Viorica Sofronie-Stokkermans. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 299–314. DOI: 10.1007/978-3-642-22438-6_23.

[39] Joe Hurd. “First-Order Proof Tactics in Higher-Order Logic Theorem Provers”. In: *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*. Ed. by Myla Archer, Ben Di Vito, and César Muñoz. NASA Technical Reports NASA/CP-2003-212448. Sept. 2003, pp. 56–68. URL: http://www.gilith.com/papers/metis.pdf.

[40] Moa Johansson, Dan Rosén, and Nicholas Smallbone. *TIP Format*. Aug. 16, 2019. URL: https://tip-org.github.io/format.html (visited on 06/24/2022).

[41] Simon L. Peyton Jones. “Haskell 98: Introduction”. In: *J. Funct. Program.* 13.1 (2003), pp. i–6. DOI: 10.1017/S0956796803000315.

[42] Michael Kohlhase et al. “Formal Management of CAD/CAM Processes”. In: *FM 2009: Formal Methods*. Ed. by Ana Cavalcanti and Dennis R. Dams. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 223–238. DOI: 10.1007/978-3-642-05089-3_15.

[43] Konstantin Korovin. “Instantiation-Based Automated Reasoning: From Theory to Practice”. In: *Automated Deduction – CADE-22*. Ed. by Renate A. Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 163–166. DOI: 10.1007/978-3-642-02959-2_14.

[44] Konstantin Korovin and Christoph Sticksel. “iProver-Eq: An Instantiation-Based Theorem Prover with Equality”. In: *Automated Reasoning*. Ed. by Jürgen Giesl and Reiner Hähnle. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 196–202. DOI: 10.1007/978-3-642-14203-1_17.

[45] Fredrik Lindblad. "A Focused Sequent Calculus for Higher-Order Logic". In: *Automated Reasoning*. Ed. by Stéphane Demri, Deepak Kapur, and Christoph Weidenbach. Cham: Springer International Publishing, 2014, pp. 61–75. DOI: 10.1007/978-3-319-08587-6_5.

[46] Christopher Menzel. "Knowledge representation, the World Wide Web, and the evolution of logic". In: *Synthese* 182.2 (Sept. 2011), pp. 269–295. DOI: 10.1007/s11229-009-9661-2.

[47] José Meseguer. "General Logics". In: *Logic Colloquium'87*. Ed. by H.-D. Ebbinghaus et al. Vol. 129. Studies in Logic and the Foundations of Mathematics. Elsevier, 1989, pp. 275–329. DOI: 10.1016/S0049-237X(08)70132-0.

[48] Eugenio Moggi. "Categories of partial morphisms and the λP-calculus". In: *Category Theory and Computer Programming: Tutorial and Workshop, Guildford, U.K. September 16–20, 1985 Proceedings*. Ed. by David Pitt et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 242–251. DOI: 10.1007/3-540-17162-2_126.

[49] Till Mossakowski. "Different types of arrow between logical frameworks". In: *Automata, Languages and Programming*. Ed. by Friedhelm Meyer and Burkhard Monien. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 158–169. DOI: 10.1007/3-540-61440-0_125.

[50] Till Mossakowski. "Relating CASL with other specification languages: the institution level". In: *Theoretical Computer Science* 286.2 (2002). Current trends in Algebraic Development Techniques, pp. 367–475. DOI: 10.1016/S0304-3975(01)00369-3.

[51] Till Mossakowski. "Foundations of Heterogeneous Specification". In: *Recent Trends in Algebraic Development Techniques*. Ed. by Martin Wirsing, Dirk Pattinson, and Rolf Hennicker. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 359–375. DOI: 10.1007/978-3-540-40020-2_21.

[52] Till Mossakowski. "Heterogeneous specification and the heterogeneous tool set". Habilitation thesis. University of Bremen, 2005. URL: https://iks.cs.ovgu.de/~till/papers/habil.pdf.

[53] Till Mossakowski et al. "Casl — the Common Algebraic Specification Language". In: *Logics of Specification Languages*. Ed. by Dines Bjørner and Martin C. Henson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 241–298. DOI: 10.1007/978-3-540-74107-7_5.

[54] Till Mossakowski et al. "Proof Support for Common Logic". In: *Automated Reasoning in Quantified Non-Classical Logics, ARQNL@IJCAR 2014, Vienna, Austria, July 23, 2014*. Ed. by Christoph Benzmüller and Jens Otten. Vol. 33. EPiC Series in Computing. EasyChair, 2014, pp. 42–58. DOI: 10.29007/2ksh.

[55] Peter D. Mosses. "Casl Syntax". In: *Casl Reference Manual. The Complete Documentation of the Common Algebraic Specification Language*. Ed. by Peter D. Mosses. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 73–112. DOI: 10.1007/978-3-540-24648-0_2.

[56] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[57] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002. DOI: 10.1007/3-540-45949-9.

[58] Visa Nummelin et al. "Superposition with First-class Booleans and Inprocessing Clausification". In: *Automated Deduction – CADE 28*. Ed. by André Platzer and Geoff Sutcliffe. Cham: Springer International Publishing, 2021, pp. 378–395. DOI: 10.1007/978-3-030-79876-5_22.

[59] S. Owre et al. "PVS: Combining specification, proof checking, and model checking". In: *Computer Aided Verification*. Ed. by Rajeev Alur and Thomas A. Henzinger. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 411–414. DOI: 10.1007/3-540-61474-5_91.

[60] Lawrence C. Paulson. "Natural deduction as higher-order resolution". In: *The Journal of Logic Programming* 3.3 (1986), pp. 237–258. DOI: 10.1016/0743-1066(86)90015-4.

[61] Lawrence C. Paulson and Kong Woei Susanto. "Source-Level Proof Reconstruction for Interactive Theorem Proving". In: *Theorem Proving in Higher Order Logics*. Ed. by Klaus Schneider and Jens Brandt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 232–245. DOI: 10.1007/978-3-540-74591-4_18.

[62] Didier Rémy. "Using, Understanding, and Unraveling the OCaml Language From Practice to Theory and Vice Versa". In: *Applied Semantics*. Ed. by Gilles Barthe et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 413–536. DOI: 10.1007/3-540-45699-6_9.

[63] Alexandre Riazanov and Andrei Voronkov. "The design and implementation of VAMPIRE". In: *AI Commun.* 15.2-3 (2002), pp. 91–110. URL: http://content.iospress.com/articles/ai-communications/aic259.

[64] Dan Rosén and Nicholas Smallbone. "TIP: Tools for Inductive Provers". In: *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*. Ed. by Martin Davis et al. Vol. 9450. Lecture Notes in Computer Science. Springer, 2015, pp. 219–232. DOI: 10.1007/978-3-662-48899-7_16.

[65] Tobias Rosenberger, Alexander Knapp, and Markus Roggenbach. "An Institutional Approach to Communicating UML State Machines". In: *Fundamental Approaches to Software Engineering*. Ed. by Einar Broch Johnsen and Manuel Wimmer. Cham: Springer International Publishing, 2022, pp. 205–224.

[66] Barkley Rosser. "Extensions of Some Theorems of Gödel and Church". In: *The Journal of Symbolic Logic* 1.3 (Sept. 1936), pp. 87–91. DOI: 10.2307/2269028.

[67] Philipp Rümmer. "A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Iliano Cervesato, Helmut Veith, and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 274–289. DOI: 10.1007/978-3-540-89439-1_20.

[68] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. "Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values". In: *Proceedings of the First ACM SIGPLAN Symposium on Haskell*. Haskell '08. Victoria, BC, Canada: Association for Computing Machinery, 2008, pp. 37–48. DOI: 10.1145/1411286.1411292.

[69] Donald Sannella and Andrzej Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-17336-3_1.

[70] Lutz Schröder. "The HASCASL prologue: Categorical syntax and semantics of the partial -calculus". In: *Theoretical Computer Science* 353.1 (2006), pp. 1–25. DOI: 10.1016/j.tcs.2005.06.037.

[71] Lutz Schröder and Till Mossakowski. "HasCasl: Towards Integrated Specification and Development of Functional Programs". In: *Algebraic Methodology and Software Technology*. Ed. by Hélène Kirchner and Christophe Ringeissen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 99–116. DOI: 10.1007/3-540-45719-4_8.

[72] Lutz Schröder and Till Mossakowski. "HasCasl: Integrated higher-order specification and program development". In: *Theoretical Computer Science* 410.12 (2009), pp. 1217–1260. DOI: 10.1016/j.tcs.2008.11.020.

[73] Stephan Schulz, Simon Cruanes, and Petar Vukmirović. "Faster, Higher, Stronger: E 2.3". In: *Automated Deduction – CADE 27*. Ed. by Pascal Fontaine. Cham: Springer International Publishing, 2019, pp. 495–507. DOI: 10.1007/978-3-030-29436-6_29.

[74] Dana Scott. "Identity and existence in intuitionistic logic". In: *Applications of Sheaves: Proceedings of the Research Symposium on Applications of Sheaf Theory to Logic, Algebra, and Analysis, Durham, July 9–21, 1977*. Ed. by Michael Fourman, Christopher Mulvey, and Dana Scott. Berlin, Heidelberg: Springer Berlin Heidelberg, 1979, pp. 660–696. DOI: 10.1007/BFb0061839.

[75] Ian Sommerville. *Software Engineering*. 10th ed. Global Edition. Pearson Deutschland, 2016. 810 pp. URL: https://elibrary.pearson.de/book/99.150005/9781292096148.

[76] Alexander Steen, Max Wisniewski, and Christoph Benzmüller. "Agent-Based HOL Reasoning". In: *Mathematical Software – ICMS 2016*. Ed. by Gert-Martin Greuel et al. Cham: Springer International Publishing, 2016, pp. 75–81. DOI: 10.1007/978-3-319-42432-3_10.

[77] Mark Stickel et al. "Deductive composition of astronomical software from subroutine libraries". In: *Automated Deduction — CADE-12*. Ed. by Alan Bundy. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 341–355. DOI: 10.1007/3-540-58156-1_24.

[78] Geoff Sutcliffe, Christian B. Suttner, and Theodor Yemenis. "The TPTP Problem Library". In: *Automated Deduction - CADE-12, 12th International Conference on Automated Deduction, Nancy, France, June 26 - July 1, 1994, Proceedings*. Ed. by Alan Bundy. Vol. 814. Lecture Notes in Computer Science. Springer, 1994, pp. 252–266. DOI: 10.1007/3-540-58156-1_18.

[79] Geoff Sutcliffe et al. “The TPTP Typed First-Order Form with Arithmetic”. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Nikolaj Bjørner and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 406–419. DOI: 10.1007/978-3-642-28717-6_32.

[80] Wouter Swierstra. “xmonad in Coq *(experience report)*: programming a window manager in a proof assistant”. In: *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012*. Ed. by Janis Voigtländer. ACM, 2012, pp. 131–136. DOI: 10.1145/2364506.2364523.

[81] Technical Committee ISO/IEC JTC 1. *Information technology – Common Logic (CL) – A framework for a family of logic-based languages*. International Standard ISO/IEC 24707. 2nd edition. July 2018. URL: https://standards.iso.org/ittf/PubliclyAvailableStandards/c066249_ISO_IEC_24707_2018.zip.

[82] Richard Uhler and Nirav Dave. “Smten: Automatic Translation of High-Level Symbolic Computations into SMT Queries”. In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 678–683. DOI: 10.1007/978-3-642-39799-8_45.

[83] Daniel Wand. “Superposition: Types and Induction”. PhD thesis. Saarland University, Aug. 2017. URL: https://hal.inria.fr/tel-01592497.

[84] Christoph Weidenbach et al. “SPASS Version 3.5”. In: *Automated Deduction – CADE-22*. Ed. by Renate A. Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 140–145. DOI: 10.1007/978-3-642-02959-2_10.

Appendix

A.1 CASL specifications for evaluation

```
library Datatypes
spec Nat =
  free type Nat ::= 0 | suc(Nat)
  ops   __ + __  :   Nat * Nat ->  Nat;
  forall m,n,k : Nat
  . 0 + m = m                          %(add_0)%         %simp
  . suc(n) + m = suc(n + m)            %(add_suc)%       %simp
  . m + 0 = m                          %(add_0_right)%   %implied %simp
  . m+(n+k) = (m+n)+k                  %(add_assoc)%     %implied %simp
  . m+suc(n) = suc(m+n)                %(add_suc_right)% %implied %simp
  . m+n = n+m                          %(add_comm)%      %implied
  pred __<=__ : Nat*Nat
  forall m,n : Nat
  . 0 <= n                             %(leq_def1)% %simp
  . not suc(n) <= 0                    %(leq_def2)% %simp
  . suc(m) <= suc(n) <=> m <= n        %(leq_def3)% %simp
  ops min,max:   Nat * Nat -> Nat
  forall m,n,k : Nat
  . min(m,n) = m when m <= n else n  %(min_def)% %simp
  . max(m,n) = n when m <= n else m  %(max_def)% %simp
end
spec List [sort Elem] given Nat =
    free type List ::= nil | __::__(Elem; List)
    ops __++__ : List * List -> List;
        reverse : List -> List;
        length : List -> Nat
    vars x:Elem; K, L, M:List
    . nil ++ K = K %(concat_nil)%                              %simp
    . (x :: K) ++ L = x :: (K ++ L) %(concat_NeList)%          %simp
    . reverse(nil) = nil %(reverse_nil)%                       %simp
    . reverse(x :: L) = reverse(L) ++ (x :: nil) %(reverse_NeList)% %simp
    . length(nil) = 0 %(length_nil)%                           %simp
    . length(x :: L) = suc(length(L)) %(length_NeList)%        %simp
    . K++(L++M) = (K++L)++M %(concat_assoc)%                   %implied %simp
    . K ++nil = K %(concat_nil_right)%                         %implied %simp
    . reverse(K ++ L) = reverse(L) ++ reverse(K) %(reverse_concat)% %implied %simp
    . length(K ++ L) = length(K) + length(L)     %(length_concat)%  %implied %simp
end
spec BinTree[sort Elem] given Nat, List[sort Elem] =
  free type Tree ::= Nil | Bin(Tree;Elem;Tree)
  op flip : Tree -> Tree
  forall x:Elem; t,u:Tree
  . flip(Nil) = Nil                              %(flip_Nil)%  %simp
  . flip(Bin(t,x,u)) = Bin(flip(u),x,flip(t))  %(flip_Bin)%  %simp
  . flip(flip(t)) = t                            %(flip_flip)% %implied
  op height : Tree -> Nat
  forall x:Elem; t,u:Tree
  . height(Nil) = 0                              %(height_Nil)% %simp
  . height(Bin(t,x,u)) = suc(max(height(t),height(u))) %(height_Bin)% %simp
  pred balanced : Tree
  forall x:Elem; t,u:Tree
  . balanced(Nil)                                %(balanced_Nil)% %simp
  . balanced(Bin(t,x,u)) <=>
      balanced(t) /\ balanced(u) /\ height(t)=height(u) %(balanced_Bin)% %simp
  op fringe : Tree -> List
  forall x:Elem; t,u:Tree
  . fringe(Nil) = nil                              %(fringe_Nil)% %simp
  . fringe(Bin(t,x,u)) = (fringe(t)++(x::nil))++fringe(u) %(fringe_Bin)% %simp
  . forall l:List . exists t:Tree . fringe(t)=l  %(balanced_existence)% %implied
end
```

Listing A.1: CASL specifications for $\mathbb{N}$, lists and binary trees, restricted to $CFOL^{=}$ features (courtesy of Till Mossakowski)

```
library Datatypes
spec Nat =
  free type Nat ::= 0 | suc(pre:? Nat)
  ops   __ + __  :   Nat * Nat ->  Nat;
  forall m,n,k : Nat
  . 0 + m = m                          %(add_0)%         %simp
  . suc(n) + m = suc(n + m)            %(add_suc)%       %simp
  . m + 0 = m                          %(add_0_right)%   %implied %simp
  . m+(n+k) = (m+n)+k                  %(add_assoc)%     %implied %simp
  . m+suc(n) = suc(m+n)                %(add_suc_right)% %implied %simp
  . m+n = n+m                          %(add_comm)%      %implied
  pred __<=__ : Nat*Nat
  forall m,n : Nat
  . 0 <= n                             %(leq_def1)% %simp
  . not suc(n) <= 0                    %(leq_def2)% %simp
  . suc(m) <= suc(n) <=> m <= n        %(leq_def3)% %simp
  ops min,max:   Nat * Nat -> Nat
  forall m,n,k : Nat
  . min(m,n) = m when m <= n else n  %(min_def)% %simp
  . max(m,n) = n when m <= n else m  %(max_def)% %simp
end
spec List [sort Elem] given Nat =
    free type List ::= nil | __::__(head:? Elem; tail:? List)
    ops __++__ : List * List -> List;
        reverse : List -> List;
        length : List -> Nat
    vars x:Elem; K, L, M:List
    . nil ++ K = K %(concat_nil)%                              %simp
    . (x :: K) ++ L = x :: (K ++ L) %(concat_NeList)%          %simp
    . reverse(nil) = nil %(reverse_nil)%                       %simp
    . reverse(x :: L) = reverse(L) ++ (x :: nil) %(reverse_NeList)% %simp
    . length(nil) = 0 %(length_nil)%                                %simp
    . length(x :: L) = suc(length(L)) %(length_NeList)%             %simp
    . K++(L++M) = (K++L)++M %(concat_assoc)%                        %implied %simp
    . K ++nil = K %(concat_nil_right)%                              %implied %simp
    . reverse(K ++ L) = reverse(L) ++ reverse(K) %(reverse_concat)% %implied %simp
    . length(K ++ L) = length(K) + length(L)     %(length_concat)%  %implied %simp
end
spec BinTree[sort Elem] given Nat, List[sort Elem] =
  free type Tree ::= Nil | Bin(left:? Tree;label:? Elem;right:? Tree)
  op flip : Tree -> Tree
  forall x:Elem; t,u:Tree
  . flip(Nil) = Nil                              %(flip_Nil)%  %simp
  . flip(Bin(t,x,u)) = Bin(flip(u),x,flip(t))  %(flip_Bin)%  %simp
  . flip(flip(t)) = t                            %(flip_flip)% %implied
  op height : Tree -> Nat
  forall x:Elem; t,u:Tree
  . height(Nil) = 0                              %(height_Nil)% %simp
  . height(Bin(t,x,u)) = suc(max(height(t),height(u))) %(height_Bin)% %simp
  pred balanced : Tree
  forall x:Elem; t,u:Tree
  . balanced(Nil)                                %(balanced_Nil)% %simp
  . balanced(Bin(t,x,u)) <=>
      balanced(t) /\ balanced(u) /\ height(t)=height(u) %(balanced_Bin)% %simp
  op fringe : Tree -> List
  forall x:Elem; t,u:Tree
  . fringe(Nil) = nil                              %(fringe_Nil)% %simp
  . fringe(Bin(t,x,u)) = (fringe(t)++(x::nil))++fringe(u) %(fringe_Bin)% %simp
  . forall l:List . exists t:Tree . fringe(t)=l  %(balanced_existence)% %implied
end
```

Listing A.2: CASL specifications from Listing A.1, extended with partial functions

```
library Datatypes
spec Nat =
  sort Pos < Nat
  free type Pos ::= 1 | suc(Pos)
  free type Nat ::= 0 | suc(Nat)
  ops   __ + __  :   Nat * Nat ->  Nat;
  forall m,n,k : Nat; p,q,r : Pos
  . suc(0) = 1                        %(Pos_def)%           %simp
  . 0 + m = m                         %(add_0)%         %simp
  . suc(n) + m = suc(n + m)           %(add_suc)%       %simp
  . p + 0 = p                         %(add_0_right)%   %implied %simp
  . p+(q+r) = (p+q)+r                 %(add_assoc)%     %implied %simp
  . p+suc(q) = suc(p+q)               %(add_suc_right)% %implied %simp
  . p+q = q+p                         %(add_comm)%      %implied
  pred __<=__ : Nat*Nat
  forall m,n : Nat
  . 0 <= n                            %(leq_def1)% %simp
  . not suc(n) <= 0                   %(leq_def2)% %simp
  . suc(m) <= suc(n) <=> m <= n       %(leq_def3)% %simp
  ops min,max:   Nat * Nat -> Nat
  forall m,n,k : Nat
  . min(m,n) = m when m <= n else n  %(min_def)% %simp
  . max(m,n) = n when m <= n else m  %(max_def)% %simp
end
spec List [sort Elem] given Nat =
    sort NEList < List
    free type NEList ::= [__](Elem) | __::__(Elem; NEList)
    free type List ::= nil | __::__(Elem; List)
    ops __++__ : List * List -> List;
        reverse : List -> List;
        length : List -> Nat
    vars x:Elem; K, L, M:List; P, Q, R:NEList
    . x :: nil = [x] %(NEList_def)%                          %simp
    . nil ++ K = K %(concat_nil)%                            %simp
    . (x :: K) ++ L = x :: (K ++ L) %(concat_NeList)%        %simp
    . reverse(nil) = nil %(reverse_nil)%                     %simp
    . reverse(x :: L) = reverse(L) ++ (x :: nil) %(reverse_NeList)% %simp
    . length(nil) = 0 %(length_nil)%                                %simp
    . length(x :: L) = suc(length(L)) %(length_NeList)%             %simp
    . P++(Q++R) = (P++Q)++R %(concat_assoc)%                        %implied %simp
    . P ++nil = P %(concat_nil_right)%                              %implied %simp
    . reverse(P ++ Q) = reverse(Q) ++ reverse(P) %(reverse_concat)% %implied %simp
    . length(P ++ Q) = length(P) + length(Q)     %(length_concat)%  %implied %simp
end
spec BinTree[sort Elem] given Nat, List[sort Elem] =
  sort NETree < Tree
  free type NETree ::= Leaf(Elem) | Left(NETree;Elem) | Right(Elem;NETree)| Bin(NETree
      ;Elem;NETree)
  free type Tree ::= Nil | Bin(Tree;Elem;Tree)
  op flip : Tree -> Tree
  forall x:Elem; t,u:Tree; v:NETree
  . Bin(Nil,x,Nil) = Leaf(x)                    %(NETree_def1)%
  . Bin(Nil,x,v) = Right(x,v)                   %(NETree_def2)%
  . Bin(v,x,Nil) = Left(v,x)                    %(NETree_def3)%
  . flip(Nil) = Nil                             %(flip_Nil)%  %simp
  . flip(Bin(t,x,u)) = Bin(flip(u),x,flip(t))  %(flip_Bin)%  %simp
  . flip(flip(v)) = v                           %(flip_flip)% %implied
  op height : Tree -> Nat
  forall x:Elem; t,u:Tree
  . height(Nil) = 0                             %(height_Nil)% %simp
  . height(Bin(t,x,u)) = suc(max(height(t),height(u))) %(height_Bin)% %simp
  pred balanced : Tree
  forall x:Elem; t,u:Tree
  . balanced(Nil)                               %(balanced_Nil)% %simp
  . balanced(Bin(t,x,u)) <=>
      balanced(t) /\ balanced(u) /\ height(t)=height(u) %(balanced_Bin)% %simp
  op fringe : Tree -> List
  forall x:Elem; t,u:Tree
  . fringe(Nil) = nil                             %(fringe_Nil)% %simp
  . fringe(Bin(t,x,u)) = (fringe(t)++(x::nil))++fringe(u) %(fringe_Bin)% %simp
  . forall l:NEList . exists t:NETree . fringe(t)=l  %(balanced_existence)% %implied
end
```

Listing A.3: CASL specifications from Listing A.1, extended with subsorting features

```
library Datatypes
spec Nat =
  sort Pos < Nat
  free type Pos ::= 1 | suc(Pos)
  free type Nat ::= 0 | suc(Nat)
  ops   __ + __  :   Nat * Nat ->  Nat;
  forall m,n,k : Nat; p,q,r : Pos
  . suc(0) = 1                        %(Pos_def)%           %simp
  . 0 + m = m                         %(add_0_Nat)%         %simp
  . suc(n) + m = suc(n + m)           %(add_suc_Nat)%       %simp
  . p + 0 = p                         %(add_0_Pos_right)%   %implied %simp
  . p+(q+r) = (p+q)+r                 %(add_assoc_Pos)%     %implied %simp
  . p+suc(q) = suc(p+q)               %(add_suc_Pos_right)% %implied %simp
  . p+q = q+p                         %(add_comm_Pos)%      %implied
  pred __<=__ : Nat*Nat
  forall m,n : Nat
  . 0 <= n                            %(leq_def1_Nat)% %simp
  . not suc(n) <= 0                   %(leq_def2_Nat)% %simp
  . suc(m) <= suc(n) <=> m <= n       %(leq_def3_Nat)% %simp
  ops min,max:   Nat * Nat -> Nat
  forall m,n,k : Nat
  . min(m,n) = m when m <= n else n  %(min_def_Nat)% %simp
  . max(m,n) = n when m <= n else m  %(max_def_Nat)% %simp
end
spec List [sort Elem] given Nat =
    sort NEList < List
    free type NEList ::= [__](Elem) | __::__(Elem; NEList)
    free type List ::= nil | __::__(Elem; List)
    ops __++__ : List * List -> List;
        reverse : List -> List;
        length : List -> Nat
    vars x:Elem; K, L, M:List; P, Q, R:NEList
    . x :: nil = [x] %(NEList_def)%                          %simp
    . nil ++ K = K %(concat_nil)%                            %simp
    . (x :: K) ++ L = x :: (K ++ L) %(concat_NeList)%        %simp
    . reverse(nil) = nil %(reverse_nil)%                     %simp
    . reverse(x :: L) = reverse(L) ++ (x :: nil) %(reverse_NeList)% %simp
    . length(nil) = 0 %(length_nil)%                                 %simp
    . length(x :: L) = suc(length(L)) %(length_NeList)%              %simp
    . P++(Q++R) = (P++Q)++R %(concat_assoc)%                         %implied %simp
    . P ++nil = P %(concat_nil_right)%                               %implied %simp
    . reverse(P ++ Q) = reverse(Q) ++ reverse(P) %(reverse_concat)% %implied %simp
    . length(P ++ Q) = length(P) + length(Q)     %(length_concat)%  %implied %simp
end
spec BinTree[sort Elem] given Nat, List[sort Elem] =
  sort NETree < Tree
  free type NETree ::= Leaf(Elem) | Left(NETree;Elem) | Right(Elem;NETree)| Bin(NETree
      ;Elem;NETree)
  free type Tree ::= Nil | Bin(Tree;Elem;Tree)
  op flip : Tree -> Tree
  forall x:Elem; t,u:Tree; v:NETree
  . Bin(Nil,x,Nil) = Leaf(x)                    %(NETree_def1)%
  . Bin(Nil,x,v) = Right(x,v)                   %(NETree_def2)%
  . Bin(v,x,Nil) = Left(v,x)                    %(NETree_def3)%
  . flip(Nil) = Nil                             %(flip_Nil)%  %simp
  . flip(Bin(t,x,u)) = Bin(flip(u),x,flip(t))  %(flip_Bin)%  %simp
  . flip(flip(v)) = v                           %(flip_flip)% %implied
  op height : Tree -> Nat
  forall x:Elem; t,u:Tree
  . height(Nil) = 0                             %(height_Nil)% %simp
  . height(Bin(t,x,u)) = suc(max(height(t),height(u))) %(height_Bin)% %simp
  pred balanced : Tree
  forall x:Elem; t,u:Tree
  . balanced(Nil)                               %(balanced_Nil)% %simp
  . balanced(Bin(t,x,u)) <=>
      balanced(t) /\ balanced(u) /\ height(t)=height(u) %(balanced_Bin)% %simp
  op fringe : Tree -> List
  forall x:Elem; t,u:Tree
  . fringe(Nil) = nil                            %(fringe_Nil)% %simp
  . fringe(Bin(t,x,u)) = (fringe(t)++(x::nil))++fringe(u) %(fringe_Bin)% %simp
  . forall l:NEList . exists t:NETree . fringe(t)=l  %(balanced_existence)% %implied
end
```

Listing A.4: CASL specifications from Listing A.1, extended with partial functions and subsorting features

A.2 TIP translations for evaluation

```
1  (declare-datatype s_Nat ((fNat_0) (fNat->Nat_suc (i1_fNat->Nat_suc s_Nat))))
2  (declare-fun pNat*Nat___<=__ (s_Nat s_Nat) Bool)
3  (declare-fun fNat*Nat->Nat___+__ (s_Nat s_Nat) s_Nat)
4  (declare-fun fNat*Nat->Nat_max (s_Nat s_Nat) s_Nat)
5  (declare-fun fNat*Nat->Nat_min (s_Nat s_Nat) s_Nat)
6  (assert :axiom ga_injective_suc (forall ((FNat_X1 s_Nat) (FNat_Y1 s_Nat)) (= (= (fNat
       ->Nat_suc FNat_X1) (fNat->Nat_suc FNat_Y1)) (= FNat_X1 FNat_Y1))))
7  (assert :axiom ga_disjoint_0_suc (forall ((FNat_Y1 s_Nat)) (not (= (fNat_0) (fNat->
       Nat_suc FNat_Y1)))))
8  (assert :axiom ga_generated_Nat (forall ((PNat_gn_P_Nat (=> s_Nat Bool))) (=> (and (@
       PNat_gn_P_Nat (fNat_0)) (forall ((FNat_y_1 s_Nat)) (=> (@ PNat_gn_P_Nat FNat_y_1)
       (@ PNat_gn_P_Nat (fNat->Nat_suc FNat_y_1))))) (forall ((FNat_x_1 s_Nat)) (@
       PNat_gn_P_Nat FNat_x_1)))))
9  (assert :axiom add_0 (forall ((FNat_m s_Nat)) (= (fNat*Nat->Nat___+__ (fNat_0) FNat_m)
        FNat_m)))
10 (assert :axiom add_suc (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (= (fNat*Nat->Nat___+__
        (fNat->Nat_suc FNat_n) FNat_m) (fNat->Nat_suc (fNat*Nat->Nat___+__ FNat_n FNat_m)
       ))))
11 (assert :axiom leq_def1 (forall ((FNat_n s_Nat)) (pNat*Nat___<=__ (fNat_0) FNat_n)))
12 (assert :axiom leq_def2 (forall ((FNat_n s_Nat)) (not (pNat*Nat___<=__ (fNat->Nat_suc
       FNat_n) (fNat_0)))))
13 (assert :axiom leq_def3 (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (= (pNat*Nat___<=__ (
       fNat->Nat_suc FNat_m) (fNat->Nat_suc FNat_n)) (pNat*Nat___<=__ FNat_m FNat_n))))
14 (assert :axiom min_def (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (= (fNat*Nat->Nat_min
       FNat_m FNat_n) (ite (pNat*Nat___<=__ FNat_m FNat_n) FNat_m FNat_n))))
15 (assert :axiom max_def (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (= (fNat*Nat->Nat_max
       FNat_m FNat_n) (ite (pNat*Nat___<=__ FNat_m FNat_n) FNat_n FNat_m))))
16 (prove :axiom add_0_right (forall ((FNat_m s_Nat)) (= (fNat*Nat->Nat___+__ FNat_m (
       fNat_0)) FNat_m)))
17 (prove :axiom add_assoc (forall ((FNat_m s_Nat) (FNat_n s_Nat) (FNat_k s_Nat)) (= (
       fNat*Nat->Nat___+__ FNat_m (fNat*Nat->Nat___+__ FNat_n FNat_k)) (fNat*Nat->Nat___+
       __ (fNat*Nat->Nat___+__ FNat_m FNat_n) FNat_k))))
18 (prove :axiom add_comm (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (= (fNat*Nat->Nat___+__
        FNat_m FNat_n) (fNat*Nat->Nat___+__ FNat_n FNat_m))))
19 (prove :axiom add_suc_right (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (= (fNat*Nat->
       Nat___+__ FNat_m (fNat->Nat_suc FNat_n)) (fNat->Nat_suc (fNat*Nat->Nat___+__
       FNat_m FNat_n)))))
```

Listing A.5: TIP translation of `spec Nat` from Listing A.1 with features not supported by Zipperposition kept for readability

```
1  (declare-datatype s_Nat ((fNat_0) (fNat_gn_bottom_Nat) (fNat->Nat_suc (i1_fNat->
       Nat_suc s_Nat))))
2  (declare-fun pNat*Nat___<=__ (s_Nat s_Nat) Bool)
3  (declare-fun pNat_gn_defined (s_Nat) Bool)
4  (declare-fun fNat*Nat->Nat___+__ (s_Nat s_Nat) s_Nat)
5  (declare-fun fNat*Nat->Nat_max (s_Nat s_Nat) s_Nat)
6  (declare-fun fNat*Nat->Nat_min (s_Nat s_Nat) s_Nat)
7  (declare-fun fNat->Nat_pre (s_Nat) s_Nat)
8  (assert :axiom ga_nonEmpty_Nat (exists ((FNat_x s_Nat)) (pNat_gn_defined FNat_x)))
9  (assert :axiom ga_notDefBottom_Nat (not (pNat_gn_defined (fNat_gn_bottom_Nat))))
10 (assert :axiom ga_strictness_0 (pNat_gn_defined (fNat_0)))
11 (assert :axiom ga_strictness___+__ (forall ((FNat_x_1 s_Nat) (FNat_x_2 s_Nat)) (= (
       pNat_gn_defined (fNat*Nat->Nat___+__ FNat_x_1 FNat_x_2)) (and (pNat_gn_defined
       FNat_x_1) (pNat_gn_defined FNat_x_2)))))
12 (assert :axiom ga_strictness_max (forall ((FNat_x_1 s_Nat) (FNat_x_2 s_Nat)) (= (
       pNat_gn_defined (fNat*Nat->Nat_max FNat_x_1 FNat_x_2)) (and (pNat_gn_defined
       FNat_x_1) (pNat_gn_defined FNat_x_2)))))
13 (assert :axiom ga_strictness_min (forall ((FNat_x_1 s_Nat) (FNat_x_2 s_Nat)) (= (
       pNat_gn_defined (fNat*Nat->Nat_min FNat_x_1 FNat_x_2)) (and (pNat_gn_defined
       FNat_x_1) (pNat_gn_defined FNat_x_2)))))
14 (assert :axiom ga_strictness_pre (forall ((FNat_x_1 s_Nat)) (=> (pNat_gn_defined (fNat
       ->Nat_pre FNat_x_1)) (pNat_gn_defined FNat_x_1))))
15 (assert :axiom ga_strictness_suc (forall ((FNat_x_1 s_Nat)) (= (pNat_gn_defined (fNat
       ->Nat_suc FNat_x_1)) (pNat_gn_defined FNat_x_1))))
16 (assert :axiom ga_predicate_strictness___<=__ (forall ((FNat_x_1 s_Nat) (FNat_x_2
       s_Nat)) (=> (pNat*Nat___<=__ FNat_x_1 FNat_x_2) (and (pNat_gn_defined FNat_x_1) (
       pNat_gn_defined FNat_x_2)))))
17 (assert :axiom ga_selector_pre (forall ((FNat_X1 s_Nat)) (=> (pNat_gn_defined FNat_X1)
        (= (fNat->Nat_pre (fNat->Nat_suc FNat_X1)) FNat_X1))))
18 (assert :axiom ga_injective_suc (forall ((FNat_X1 s_Nat) (FNat_Y1 s_Nat)) (=> (and (
       pNat_gn_defined FNat_X1) (pNat_gn_defined FNat_Y1)) (= (= (fNat->Nat_suc FNat_X1)
       (fNat->Nat_suc FNat_Y1)) (= FNat_X1 FNat_Y1)))))
19 (assert :axiom ga_disjoint_0_suc (forall ((FNat_Y1 s_Nat)) (=> (pNat_gn_defined
       FNat_Y1) (not (= (fNat_0) (fNat->Nat_suc FNat_Y1))))))
20 (assert :axiom ga_selector_undef_pre_0 (not (pNat_gn_defined (fNat->Nat_pre (fNat_0)))
       ))
21 (assert :axiom ga_generated_Nat (forall ((PNat_gn_P_Nat (=> s_Nat Bool))) (=> (and (@
       PNat_gn_P_Nat (fNat_0)) (@ PNat_gn_P_Nat (fNat_gn_bottom_Nat)) (forall ((FNat_y_1
       s_Nat)) (=> (@ PNat_gn_P_Nat FNat_y_1) (@ PNat_gn_P_Nat (fNat->Nat_suc FNat_y_1)))
       ) (forall ((FNat_x_1 s_Nat)) (@ PNat_gn_P_Nat FNat_x_1)))))
22 (assert :axiom add_0 (forall ((FNat_m s_Nat)) (=> (pNat_gn_defined FNat_m) (= (fNat*
       Nat->Nat___+__ (fNat_0) FNat_m) FNat_m))))
23 (assert :axiom add_suc (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (fNat*Nat->Nat___+__ (fNat->
       Nat_suc FNat_n) FNat_m) (fNat->Nat_suc (fNat*Nat->Nat___+__ FNat_n FNat_m))))))
24 (assert :axiom leq_def1 (forall ((FNat_n s_Nat)) (=> (pNat_gn_defined FNat_n) (pNat*
       Nat___<=__ (fNat_0) FNat_n))))
25 (assert :axiom leq_def2 (forall ((FNat_n s_Nat)) (=> (pNat_gn_defined FNat_n) (not (
       pNat*Nat___<=__ (fNat->Nat_suc FNat_n) (fNat_0))))))
26 (assert :axiom leq_def3 (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (pNat*Nat___<=__ (fNat->
       Nat_suc FNat_m) (fNat->Nat_suc FNat_n)) (pNat*Nat___<=__ FNat_m FNat_n)))))
27 (assert :axiom min_def (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (fNat*Nat->Nat_min FNat_m
       FNat_n) (ite (pNat*Nat___<=__ FNat_m FNat_n) FNat_m FNat_n)))))
28 (assert :axiom max_def (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (fNat*Nat->Nat_max FNat_m
       FNat_n) (ite (pNat*Nat___<=__ FNat_m FNat_n) FNat_n FNat_m)))))
29 (prove :axiom add_0_right (forall ((FNat_m s_Nat)) (=> (pNat_gn_defined FNat_m) (= (
       fNat*Nat->Nat___+__ FNat_m (fNat_0)) FNat_m))))
30 (prove :axiom add_assoc (forall ((FNat_m s_Nat) (FNat_n s_Nat) (FNat_k s_Nat)) (=> (
       and (pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n) (pNat_gn_defined FNat_k)) (=
        (fNat*Nat->Nat___+__ FNat_m (fNat*Nat->Nat___+__ FNat_n FNat_k)) (fNat*Nat->
       Nat___+__ (fNat*Nat->Nat___+__ FNat_m FNat_n) FNat_k)))))
31 (prove :axiom add_comm (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (fNat*Nat->Nat___+__ FNat_m
       FNat_n) (fNat*Nat->Nat___+__ FNat_n FNat_m)))))
32 (prove :axiom add_suc_right (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (fNat*Nat->Nat___+__ FNat_m (
       fNat->Nat_suc FNat_n)) (fNat->Nat_suc (fNat*Nat->Nat___+__ FNat_m FNat_n))))))
```

Listing A.6: TIP translation of `spec Nat` from Listing A.2 using a persistently liberal comorphism with features not supported by Zipperposition kept for readability

```
1  (declare-sort s_Nat 0)
2  (declare-fun pNat*Nat___<=__ (s_Nat s_Nat) Bool)
3  (declare-fun pNat_gn_defined (s_Nat) Bool)
4  (declare-const fNat_0 s_Nat)
5  (declare-fun fNat*Nat->Nat___+__ (s_Nat s_Nat) s_Nat)
6  (declare-const fNat_gn_bottom_Nat s_Nat)
7  (declare-fun fNat*Nat->Nat_max (s_Nat s_Nat) s_Nat)
8  (declare-fun fNat*Nat->Nat_min (s_Nat s_Nat) s_Nat)
9  (declare-fun fNat->Nat_pre (s_Nat) s_Nat)
10 (declare-fun fNat->Nat_suc (s_Nat) s_Nat)
11 (assert :axiom ga_nonEmpty_Nat (exists ((FNat_x s_Nat)) (pNat_gn_defined FNat_x)))
12 (assert :axiom ga_notDefBottom_Nat (forall ((FNat_x s_Nat)) (= (not (pNat_gn_defined
       FNat_x)) (= FNat_x (fNat_gn_bottom_Nat)))))
13 (assert :axiom ga_strictness_0 (pNat_gn_defined (fNat_0)))
14 (assert :axiom ga_strictness___+__ (forall ((FNat_x_1 s_Nat) (FNat_x_2 s_Nat)) (= (
       pNat_gn_defined (fNat*Nat->Nat___+__ FNat_x_1 FNat_x_2)) (and (pNat_gn_defined
       FNat_x_1) (pNat_gn_defined FNat_x_2)))))
15 (assert :axiom ga_strictness_max (forall ((FNat_x_1 s_Nat) (FNat_x_2 s_Nat)) (= (
       pNat_gn_defined (fNat*Nat->Nat_max FNat_x_1 FNat_x_2)) (and (pNat_gn_defined
       FNat_x_1) (pNat_gn_defined FNat_x_2)))))
16 (assert :axiom ga_strictness_min (forall ((FNat_x_1 s_Nat) (FNat_x_2 s_Nat)) (= (
       pNat_gn_defined (fNat*Nat->Nat_min FNat_x_1 FNat_x_2)) (and (pNat_gn_defined
       FNat_x_1) (pNat_gn_defined FNat_x_2)))))
17 (assert :axiom ga_strictness_pre (forall ((FNat_x_1 s_Nat)) (=> (pNat_gn_defined (fNat
       ->Nat_pre FNat_x_1)) (pNat_gn_defined FNat_x_1))))
18 (assert :axiom ga_strictness_suc (forall ((FNat_x_1 s_Nat)) (= (pNat_gn_defined (fNat
       ->Nat_suc FNat_x_1)) (pNat_gn_defined FNat_x_1))))
19 (assert :axiom ga_predicate_strictness___<=__ (forall ((FNat_x_1 s_Nat) (FNat_x_2
       s_Nat)) (=> (pNat*Nat___<=__ FNat_x_1 FNat_x_2) (and (pNat_gn_defined FNat_x_1) (
       pNat_gn_defined FNat_x_2)))))
20 (assert :axiom ga_selector_pre (forall ((FNat_X1 s_Nat)) (=> (pNat_gn_defined FNat_X1)
        (= (fNat->Nat_pre (fNat->Nat_suc FNat_X1)) FNat_X1))))
21 (assert :axiom ga_injective_suc (forall ((FNat_X1 s_Nat) (FNat_Y1 s_Nat)) (=> (and (
       pNat_gn_defined FNat_X1) (pNat_gn_defined FNat_Y1)) (= (= (fNat->Nat_suc FNat_X1)
       (fNat->Nat_suc FNat_Y1)) (= FNat_X1 FNat_Y1)))))
22 (assert :axiom ga_disjoint_0_suc (forall ((FNat_Y1 s_Nat)) (=> (pNat_gn_defined
       FNat_Y1) (not (= (fNat_0) (fNat->Nat_suc FNat_Y1))))))
23 (assert :axiom ga_selector_undef_pre_0 (not (pNat_gn_defined (fNat->Nat_pre (fNat_0)))
       ))
24 (assert :axiom ga_generated_Nat (forall ((PNat_gn_P_Nat (=> s_Nat Bool))) (=> (and (@
       PNat_gn_P_Nat (fNat_0)) (@ PNat_gn_P_Nat (fNat_gn_bottom_Nat)) (forall ((FNat_y_1
       s_Nat)) (=> (@ PNat_gn_P_Nat FNat_y_1) (@ PNat_gn_P_Nat (fNat->Nat_suc FNat_y_1)))
       )) (forall ((FNat_x_1 s_Nat)) (@ PNat_gn_P_Nat FNat_x_1)))))
25 (assert :axiom add_0 (forall ((FNat_m s_Nat)) (=> (pNat_gn_defined FNat_m) (= (fNat*
       Nat->Nat___+__ (fNat_0) FNat_m) FNat_m))))
26 (assert :axiom add_suc (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (fNat*Nat->Nat___+__ (fNat->
       Nat_suc FNat_n) FNat_m) (fNat->Nat_suc (fNat*Nat->Nat___+__ FNat_n FNat_m))))))
27 (assert :axiom leq_def1 (forall ((FNat_n s_Nat)) (=> (pNat_gn_defined FNat_n) (pNat*
       Nat___<=__ (fNat_0) FNat_n))))
28 (assert :axiom leq_def2 (forall ((FNat_n s_Nat)) (=> (pNat_gn_defined FNat_n) (not (
       pNat*Nat___<=__ (fNat->Nat_suc FNat_n) (fNat_0))))))
29 (assert :axiom leq_def3 (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (pNat*Nat___<=__ (fNat->
       Nat_suc FNat_m) (fNat->Nat_suc FNat_n)) (pNat*Nat___<=__ FNat_m FNat_n)))))
30 (assert :axiom min_def (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (fNat*Nat->Nat_min FNat_m
       FNat_n) (ite (pNat*Nat___<=__ FNat_m FNat_n) FNat_m FNat_n)))))
31 (assert :axiom max_def (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (fNat*Nat->Nat_max FNat_m
       FNat_n) (ite (pNat*Nat___<=__ FNat_m FNat_n) FNat_n FNat_m)))))
32 (prove :axiom add_0_right (forall ((FNat_m s_Nat)) (=> (pNat_gn_defined FNat_m) (= (
       fNat*Nat->Nat___+__ FNat_m (fNat_0)) FNat_m))))
33 (prove :axiom add_assoc (forall ((FNat_m s_Nat) (FNat_n s_Nat) (FNat_k s_Nat)) (=> (
       and (pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n) (pNat_gn_defined FNat_k)) (=
        (fNat*Nat->Nat___+__ FNat_m (fNat*Nat->Nat___+__ FNat_n FNat_k)) (fNat*Nat->
       Nat___+__ (fNat*Nat->Nat___+__ FNat_m FNat_n) FNat_k)))))
34 (prove :axiom add_comm (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (fNat*Nat->Nat___+__ FNat_m
       FNat_n) (fNat*Nat->Nat___+__ FNat_n FNat_m)))))
35 (prove :axiom add_suc_right (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (fNat*Nat->Nat___+__ FNat_m (
       fNat->Nat_suc FNat_n)) (fNat->Nat_suc (fNat*Nat->Nat___+__ FNat_m FNat_n))))))
```

Listing A.7: TIP translation of `spec Nat` from Listing A.2 using a non-persistently liberal comorphism with features not supported by Zipperposition kept for readability

```
1  (declare-datatypes ((s_Pos 0) (s_Nat 0)) (((fPos_1) (fPos_gn_bottom_Pos) (fPos->
       Pos_suc (i1_fPos->Pos_suc s_Pos))) ((fNat_0) (fNat_gn_bottom_Nat) (fNat->Nat_suc (
       i1_fNat->Nat_suc s_Nat)))))
2  (declare-fun pNat*Nat___<=__ (s_Nat s_Nat) Bool)
3  (declare-fun pNat_gn_defined (s_Nat) Bool)
4  (declare-fun pPos_gn_defined (s_Pos) Bool)
5  (declare-fun fNat*Nat->Nat___+__ (s_Nat s_Nat) s_Nat)
6  (declare-fun fPos->Nat_gn_inj_Pos_Nat (s_Pos) s_Nat)
7  (declare-fun fNat->Pos_gn_proj_Nat_Pos (s_Nat) s_Pos)
8  (declare-fun fNat*Nat->Nat_max (s_Nat s_Nat) s_Nat)
9  (declare-fun fNat*Nat->Nat_min (s_Nat s_Nat) s_Nat)
10 (assert :axiom ga_nonEmpty_Nat (exists ((FNat_x s_Nat)) (pNat_gn_defined FNat_x)))
11 (assert :axiom ga_notDefBottom_Nat (not (pNat_gn_defined (fNat_gn_bottom_Nat))))
12 (assert :axiom ga_nonEmpty_Pos (exists ((FPos_x s_Pos)) (pPos_gn_defined FPos_x)))
13 (assert :axiom ga_notDefBottom_Pos (not (pPos_gn_defined (fPos_gn_bottom_Pos))))
14 (assert :axiom ga_strictness_0 (pNat_gn_defined (fNat_0)))
15 (assert :axiom ga_strictness_1 (pPos_gn_defined (fPos_1)))
16 (assert :axiom ga_strictness___+__ (forall ((FNat_x_1 s_Nat) (FNat_x_2 s_Nat)) (= (
       pNat_gn_defined (fNat*Nat->Nat___+__ FNat_x_1 FNat_x_2)) (and (pNat_gn_defined
       FNat_x_1) (pNat_gn_defined FNat_x_2)))))
17 (assert :axiom ga_strictness_gn_inj_Pos_Nat (forall ((FPos_x_1 s_Pos)) (= (
       pNat_gn_defined (fPos->Nat_gn_inj_Pos_Nat FPos_x_1)) (pPos_gn_defined FPos_x_1))))
18 (assert :axiom ga_strictness_gn_proj_Nat_Pos (forall ((FNat_x_1 s_Nat)) (=> (
       pPos_gn_defined (fNat->Pos_gn_proj_Nat_Pos FNat_x_1)) (pNat_gn_defined FNat_x_1)))
       )
19 (assert :axiom ga_strictness_max (forall ((FNat_x_1 s_Nat) (FNat_x_2 s_Nat)) (= (
       pNat_gn_defined (fNat*Nat->Nat_max FNat_x_1 FNat_x_2)) (and (pNat_gn_defined
       FNat_x_1) (pNat_gn_defined FNat_x_2)))))
20 (assert :axiom ga_strictness_min (forall ((FNat_x_1 s_Nat) (FNat_x_2 s_Nat)) (= (
       pNat_gn_defined (fNat*Nat->Nat_min FNat_x_1 FNat_x_2)) (and (pNat_gn_defined
       FNat_x_1) (pNat_gn_defined FNat_x_2)))))
21 (assert :axiom ga_strictness_suc (forall ((FNat_x_1 s_Nat)) (= (pNat_gn_defined (fNat
       ->Nat_suc FNat_x_1)) (pNat_gn_defined FNat_x_1))))
22 (assert :axiom ga_strictness_suc_1 (forall ((FPos_x_1 s_Pos)) (= (pPos_gn_defined (
       fPos->Pos_suc FPos_x_1)) (pPos_gn_defined FPos_x_1))))
23 (assert :axiom ga_predicate_strictness___<=__ (forall ((FNat_x_1 s_Nat) (FNat_x_2
       s_Nat)) (=> (pNat*Nat___<=__ FNat_x_1 FNat_x_2) (and (pNat_gn_defined FNat_x_1) (
       pNat_gn_defined FNat_x_2)))))
24 (assert :axiom ga_function_monotonicity (forall ((FPos_x1 s_Pos)) (=> (pPos_gn_defined
        FPos_x1) (= (fNat->Nat_suc (fPos->Nat_gn_inj_Pos_Nat FPos_x1)) (fPos->
       Nat_gn_inj_Pos_Nat (fPos->Pos_suc FPos_x1))))))
25 (assert :axiom ga_embedding_injectivity_Pos_to_Nat (forall ((FPos_x s_Pos) (FPos_y
       s_Pos)) (=> (and (pPos_gn_defined FPos_x) (pPos_gn_defined FPos_y)) (=> (and (= (
       fPos->Nat_gn_inj_Pos_Nat FPos_x) (fPos->Nat_gn_inj_Pos_Nat FPos_y)) (
       pNat_gn_defined (fPos->Nat_gn_inj_Pos_Nat FPos_x))) (and (= FPos_x FPos_y) (
       pPos_gn_defined FPos_x))))))
26 (assert :axiom ga_projection_injectivity_Nat_to_Pos (forall ((FNat_x s_Nat) (FNat_y
       s_Nat)) (=> (and (pNat_gn_defined FNat_x) (pNat_gn_defined FNat_y)) (=> (and (= (
       fNat->Pos_gn_proj_Nat_Pos FNat_x) (fNat->Pos_gn_proj_Nat_Pos FNat_y)) (
       pPos_gn_defined (fNat->Pos_gn_proj_Nat_Pos FNat_x))) (and (= FNat_x FNat_y) (
       pNat_gn_defined FNat_x))))))
27 (assert :axiom ga_projection_Nat_to_Pos (forall ((FPos_x s_Pos)) (=> (pPos_gn_defined
       FPos_x) (and (= (fNat->Pos_gn_proj_Nat_Pos (fPos->Nat_gn_inj_Pos_Nat FPos_x))
       FPos_x) (pPos_gn_defined (fNat->Pos_gn_proj_Nat_Pos (fPos->Nat_gn_inj_Pos_Nat
       FPos_x)))))))
28 (assert :axiom ga_injective_suc (forall ((FPos_X1 s_Pos) (FPos_Y1 s_Pos)) (=> (and (
       pPos_gn_defined FPos_X1) (pPos_gn_defined FPos_Y1)) (= (= (fPos->Pos_suc FPos_X1)
       (fPos->Pos_suc FPos_Y1)) (= FPos_X1 FPos_Y1)))))
29 (assert :axiom ga_disjoint_1_suc (forall ((FPos_Y1 s_Pos)) (=> (pPos_gn_defined
       FPos_Y1) (not (= (fPos_1) (fPos->Pos_suc FPos_Y1))))))
30 (assert :axiom ga_generated_Pos (forall ((PPos_gn_P_Pos (=> s_Pos Bool))) (=> (and (@
       PPos_gn_P_Pos (fPos_1)) (@ PPos_gn_P_Pos (fPos_gn_bottom_Pos)) (forall ((FPos_y_1
       s_Pos)) (=> (@ PPos_gn_P_Pos FPos_y_1) (@ PPos_gn_P_Pos (fPos->Pos_suc FPos_y_1)))
       )) (forall ((FPos_x_1 s_Pos)) (@ PPos_gn_P_Pos FPos_x_1)))))
31 (assert :axiom ga_injective_suc_1 (forall ((FNat_X1 s_Nat) (FNat_Y1 s_Nat)) (=> (and (
       pNat_gn_defined FNat_X1) (pNat_gn_defined FNat_Y1)) (= (= (fNat->Nat_suc FNat_X1)
       (fNat->Nat_suc FNat_Y1)) (= FNat_X1 FNat_Y1)))))
32 (assert :axiom ga_disjoint_0_suc (forall ((FNat_Y1 s_Nat)) (=> (pNat_gn_defined
       FNat_Y1) (not (= (fNat_0) (fNat->Nat_suc FNat_Y1))))))
33 (assert :axiom ga_generated_Nat (forall ((PNat_gn_P_Nat (=> s_Nat Bool))) (=> (and (@
       PNat_gn_P_Nat (fNat_0)) (@ PNat_gn_P_Nat (fNat_gn_bottom_Nat)) (forall ((FNat_y_1
       s_Nat)) (=> (@ PNat_gn_P_Nat FNat_y_1) (@ PNat_gn_P_Nat (fNat->Nat_suc FNat_y_1)))
       )) (forall ((FNat_x_1 s_Nat)) (@ PNat_gn_P_Nat FNat_x_1)))))
```

```
34 (assert :axiom Pos_def (= (fNat->Nat_suc (fNat_0)) (fPos->Nat_gn_inj_Pos_Nat (fPos_1))
       ))
35 (assert :axiom add_0 (forall ((FNat_m s_Nat)) (=> (pNat_gn_defined FNat_m) (= (fNat*
       Nat->Nat___+__ (fNat_0) FNat_m) FNat_m))))
36 (assert :axiom add_suc (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (fNat*Nat->Nat___+__ (fNat->
       Nat_suc FNat_n) FNat_m) (fNat->Nat_suc (fNat*Nat->Nat___+__ FNat_n FNat_m))))))
37 (assert :axiom leq_def1 (forall ((FNat_n s_Nat)) (=> (pNat_gn_defined FNat_n) (pNat*
       Nat___<=__ (fNat_0) FNat_n))))
38 (assert :axiom leq_def2 (forall ((FNat_n s_Nat)) (=> (pNat_gn_defined FNat_n) (not (
       pNat*Nat___<=__ (fNat->Nat_suc FNat_n) (fNat_0))))))
39 (assert :axiom leq_def3 (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (pNat*Nat___<=__ (fNat->
       Nat_suc FNat_m) (fNat->Nat_suc FNat_n)) (pNat*Nat___<=__ FNat_m FNat_n)))))
40 (assert :axiom min_def (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (fNat*Nat->Nat_min FNat_m
       FNat_n) (ite (pNat*Nat___<=__ FNat_m FNat_n) FNat_m FNat_n)))))
41 (assert :axiom max_def (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (fNat*Nat->Nat_max FNat_m
       FNat_n) (ite (pNat*Nat___<=__ FNat_m FNat_n) FNat_n FNat_m)))))
42 (prove :axiom add_0_right (forall ((FPos_p s_Pos)) (=> (pPos_gn_defined FPos_p) (= (
       fNat*Nat->Nat___+__ (fPos->Nat_gn_inj_Pos_Nat FPos_p) (fNat_0)) (fPos->
       Nat_gn_inj_Pos_Nat FPos_p)))))
43 (prove :axiom add_assoc (forall ((FPos_p s_Pos) (FPos_q s_Pos) (FPos_r s_Pos)) (=> (
       and (pPos_gn_defined FPos_p) (pPos_gn_defined FPos_q) (pPos_gn_defined FPos_r)) (=
        (fNat*Nat->Nat___+__ (fPos->Nat_gn_inj_Pos_Nat FPos_p) (fNat*Nat->Nat___+__ (fPos
       ->Nat_gn_inj_Pos_Nat FPos_q) (fPos->Nat_gn_inj_Pos_Nat FPos_r))) (fNat*Nat->Nat___
       +__ (fNat*Nat->Nat___+__ (fPos->Nat_gn_inj_Pos_Nat FPos_p) (fPos->
       Nat_gn_inj_Pos_Nat FPos_q)) (fPos->Nat_gn_inj_Pos_Nat FPos_r))))))
44 (prove :axiom add_comm (forall ((FPos_p s_Pos) (FPos_q s_Pos)) (=> (and (
       pPos_gn_defined FPos_p) (pPos_gn_defined FPos_q)) (= (fNat*Nat->Nat___+__ (fPos->
       Nat_gn_inj_Pos_Nat FPos_p) (fPos->Nat_gn_inj_Pos_Nat FPos_q)) (fNat*Nat->Nat___+__
        (fPos->Nat_gn_inj_Pos_Nat FPos_q) (fPos->Nat_gn_inj_Pos_Nat FPos_p))))))
45 (prove :axiom add_suc_right (forall ((FPos_p s_Pos) (FPos_q s_Pos)) (=> (and (
       pPos_gn_defined FPos_p) (pPos_gn_defined FPos_q)) (= (fNat*Nat->Nat___+__ (fPos->
       Nat_gn_inj_Pos_Nat FPos_p) (fPos->Nat_gn_inj_Pos_Nat (fPos->Pos_suc FPos_q))) (
       fNat->Nat_suc (fNat*Nat->Nat___+__ (fPos->Nat_gn_inj_Pos_Nat FPos_p) (fPos->
       Nat_gn_inj_Pos_Nat FPos_q)))))))
```

Listing A.8: TIP translation of `spec Nat` from Listing A.3 using a persistently liberal comorphism with features not supported by Zipperposition kept for readability

```
1  (declare-sort s_Nat 0)
2  (declare-sort s_Pos 0)
3  (declare-fun pNat*Nat___<=__ (s_Nat s_Nat) Bool)
4  (declare-fun pNat_gn_defined (s_Nat) Bool)
5  (declare-fun pPos_gn_defined (s_Pos) Bool)
6  (declare-const fNat_0 s_Nat)
7  (declare-const fPos_1 s_Pos)
8  (declare-fun fNat*Nat->Nat___+__ (s_Nat s_Nat) s_Nat)
9  (declare-const fNat_gn_bottom_Nat s_Nat)
10 (declare-const fPos_gn_bottom_Pos s_Pos)
11 (declare-fun fPos->Nat_gn_inj_Pos_Nat (s_Pos) s_Nat)
12 (declare-fun fNat->Pos_gn_proj_Nat_Pos (s_Nat) s_Pos)
13 (declare-fun fNat*Nat->Nat_max (s_Nat s_Nat) s_Nat)
14 (declare-fun fNat*Nat->Nat_min (s_Nat s_Nat) s_Nat)
15 (declare-fun fNat->Nat_suc (s_Nat) s_Nat)
16 (declare-fun fPos->Pos_suc (s_Pos) s_Pos)
17 (assert :axiom ga_nonEmpty_Nat (exists ((FNat_x s_Nat)) (pNat_gn_defined FNat_x)))
18 (assert :axiom ga_notDefBottom_Nat (forall ((FNat_x s_Nat)) (= (not (pNat_gn_defined
       FNat_x)) (= FNat_x (fNat_gn_bottom_Nat)))))
19 (assert :axiom ga_nonEmpty_Pos (exists ((FPos_x s_Pos)) (pPos_gn_defined FPos_x)))
20 (assert :axiom ga_notDefBottom_Pos (forall ((FPos_x s_Pos)) (= (not (pPos_gn_defined
       FPos_x)) (= FPos_x (fPos_gn_bottom_Pos)))))
21 (assert :axiom ga_strictness_0 (pNat_gn_defined (fNat_0)))
22 (assert :axiom ga_strictness_1 (pPos_gn_defined (fPos_1)))
23 (assert :axiom ga_strictness___+__ (forall ((FNat_x_1 s_Nat) (FNat_x_2 s_Nat)) (= (
       pNat_gn_defined (fNat*Nat->Nat___+__ FNat_x_1 FNat_x_2)) (and (pNat_gn_defined
       FNat_x_1) (pNat_gn_defined FNat_x_2)))))
24 (assert :axiom ga_strictness_gn_inj_Pos_Nat (forall ((FPos_x_1 s_Pos)) (= (
       pNat_gn_defined (fPos->Nat_gn_inj_Pos_Nat FPos_x_1)) (pPos_gn_defined FPos_x_1))))
25 (assert :axiom ga_strictness_gn_proj_Nat_Pos (forall ((FNat_x_1 s_Nat)) (=> (
       pPos_gn_defined (fNat->Pos_gn_proj_Nat_Pos FNat_x_1)) (pNat_gn_defined FNat_x_1)))
       )
26 (assert :axiom ga_strictness_max (forall ((FNat_x_1 s_Nat) (FNat_x_2 s_Nat)) (= (
       pNat_gn_defined (fNat*Nat->Nat_max FNat_x_1 FNat_x_2)) (and (pNat_gn_defined
       FNat_x_1) (pNat_gn_defined FNat_x_2)))))
27 (assert :axiom ga_strictness_min (forall ((FNat_x_1 s_Nat) (FNat_x_2 s_Nat)) (= (
       pNat_gn_defined (fNat*Nat->Nat_min FNat_x_1 FNat_x_2)) (and (pNat_gn_defined
       FNat_x_1) (pNat_gn_defined FNat_x_2)))))
28 (assert :axiom ga_strictness_suc (forall ((FNat_x_1 s_Nat)) (= (pNat_gn_defined (fNat
       ->Nat_suc FNat_x_1)) (pNat_gn_defined FNat_x_1))))
29 (assert :axiom ga_strictness_suc_1 (forall ((FPos_x_1 s_Pos)) (= (pPos_gn_defined (
       fPos->Pos_suc FPos_x_1)) (pPos_gn_defined FPos_x_1))))
30 (assert :axiom ga_predicate_strictness___<=__ (forall ((FNat_x_1 s_Nat) (FNat_x_2
       s_Nat)) (=> (pNat*Nat___<=__ FNat_x_1 FNat_x_2) (and (pNat_gn_defined FNat_x_1) (
       pNat_gn_defined FNat_x_2)))))
31 (assert :axiom ga_function_monotonicity (forall ((FPos_x1 s_Pos)) (=> (pPos_gn_defined
        FPos_x1) (= (fNat->Nat_suc (fPos->Nat_gn_inj_Pos_Nat FPos_x1)) (fPos->
       Nat_gn_inj_Pos_Nat (fPos->Pos_suc FPos_x1))))))
32 (assert :axiom ga_embedding_injectivity_Pos_to_Nat (forall ((FPos_x s_Pos) (FPos_y
       s_Pos)) (=> (and (pPos_gn_defined FPos_x) (pPos_gn_defined FPos_y)) (=> (and (= (
       fPos->Nat_gn_inj_Pos_Nat FPos_x) (fPos->Nat_gn_inj_Pos_Nat FPos_y)) (
       pNat_gn_defined (fPos->Nat_gn_inj_Pos_Nat FPos_x))) (and (= FPos_x FPos_y) (
       pPos_gn_defined FPos_x))))))
33 (assert :axiom ga_projection_injectivity_Nat_to_Pos (forall ((FNat_x s_Nat) (FNat_y
       s_Nat)) (=> (and (pNat_gn_defined FNat_x) (pNat_gn_defined FNat_y)) (=> (and (= (
       fNat->Pos_gn_proj_Nat_Pos FNat_x) (fNat->Pos_gn_proj_Nat_Pos FNat_y)) (
       pPos_gn_defined (fNat->Pos_gn_proj_Nat_Pos FNat_x))) (and (= FNat_x FNat_y) (
       pNat_gn_defined FNat_x))))))
34 (assert :axiom ga_projection_Nat_to_Pos (forall ((FPos_x s_Pos)) (=> (pPos_gn_defined
       FPos_x) (and (= (fNat->Pos_gn_proj_Nat_Pos (fPos->Nat_gn_inj_Pos_Nat FPos_x))
       FPos_x) (pPos_gn_defined (fNat->Pos_gn_proj_Nat_Pos (fPos->Nat_gn_inj_Pos_Nat
       FPos_x)))))))
35 (assert :axiom ga_injective_suc (forall ((FPos_X1 s_Pos) (FPos_Y1 s_Pos)) (=> (and (
       pPos_gn_defined FPos_X1) (pPos_gn_defined FPos_Y1)) (= (= (fPos->Pos_suc FPos_X1)
       (fPos->Pos_suc FPos_Y1)) (= FPos_X1 FPos_Y1)))))
36 (assert :axiom ga_disjoint_1_suc (forall ((FPos_Y1 s_Pos)) (=> (pPos_gn_defined
       FPos_Y1) (not (= (fPos_1) (fPos->Pos_suc FPos_Y1))))))
37 (assert :axiom ga_generated_Pos (forall ((PPos_gn_P_Pos (=> s_Pos Bool))) (=> (and (@
       PPos_gn_P_Pos (fPos_1)) (@ PPos_gn_P_Pos (fPos_gn_bottom_Pos)) (forall ((FPos_y_1
       s_Pos)) (=> (@ PPos_gn_P_Pos FPos_y_1) (@ PPos_gn_P_Pos (fPos->Pos_suc FPos_y_1)))
       )) (forall ((FPos_x_1 s_Pos)) (@ PPos_gn_P_Pos FPos_x_1)))))
38 (assert :axiom ga_injective_suc_1 (forall ((FNat_X1 s_Nat) (FNat_Y1 s_Nat)) (=> (and (
       pNat_gn_defined FNat_X1) (pNat_gn_defined FNat_Y1)) (= (= (fNat->Nat_suc FNat_X1)
       (fNat->Nat_suc FNat_Y1)) (= FNat_X1 FNat_Y1)))))
```

```
39 (assert :axiom ga_disjoint_0_suc (forall ((FNat_Y1 s_Nat)) (=> (pNat_gn_defined
       FNat_Y1) (not (= (fNat_0) (fNat->Nat_suc FNat_Y1))))))
40 (assert :axiom ga_generated_Nat (forall ((PNat_gn_P_Nat (=> s_Nat Bool))) (=> (and (@
       PNat_gn_P_Nat (fNat_0)) (@ PNat_gn_P_Nat (fNat_gn_bottom_Nat)) (forall ((FNat_y_1
       s_Nat)) (=> (@ PNat_gn_P_Nat FNat_y_1) (@ PNat_gn_P_Nat (fNat->Nat_suc FNat_y_1)))
       )) (forall ((FNat_x_1 s_Nat)) (@ PNat_gn_P_Nat FNat_x_1)))))
41 (assert :axiom Pos_def (= (fNat->Nat_suc (fNat_0)) (fPos->Nat_gn_inj_Pos_Nat (fPos_1))
       ))
42 (assert :axiom add_0 (forall ((FNat_m s_Nat)) (=> (pNat_gn_defined FNat_m) (= (fNat*
       Nat->Nat___+__ (fNat_0) FNat_m) FNat_m))))
43 (assert :axiom add_suc (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (fNat*Nat->Nat___+__ (fNat->
       Nat_suc FNat_n) FNat_m) (fNat->Nat_suc (fNat*Nat->Nat___+__ FNat_n FNat_m))))))
44 (assert :axiom leq_def1 (forall ((FNat_n s_Nat)) (=> (pNat_gn_defined FNat_n) (pNat*
       Nat___<=__ (fNat_0) FNat_n))))
45 (assert :axiom leq_def2 (forall ((FNat_n s_Nat)) (=> (pNat_gn_defined FNat_n) (not (
       pNat*Nat___<=__ (fNat->Nat_suc FNat_n) (fNat_0))))))
46 (assert :axiom leq_def3 (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (pNat*Nat___<=__ (fNat->
       Nat_suc FNat_m) (fNat->Nat_suc FNat_n)) (pNat*Nat___<=__ FNat_m FNat_n)))))
47 (assert :axiom min_def (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (fNat*Nat->Nat_min FNat_m
       FNat_n) (ite (pNat*Nat___<=__ FNat_m FNat_n) FNat_m FNat_n)))))
48 (assert :axiom max_def (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (fNat*Nat->Nat_max FNat_m
       FNat_n) (ite (pNat*Nat___<=__ FNat_m FNat_n) FNat_n FNat_m)))))
49 (prove :axiom add_0_right (forall ((FPos_p s_Pos)) (=> (pPos_gn_defined FPos_p) (= (
       fNat*Nat->Nat___+__ (fPos->Nat_gn_inj_Pos_Nat FPos_p) (fNat_0)) (fPos->
       Nat_gn_inj_Pos_Nat FPos_p)))))
50 (prove :axiom add_assoc (forall ((FPos_p s_Pos) (FPos_q s_Pos) (FPos_r s_Pos)) (=> (
       and (pPos_gn_defined FPos_p) (pPos_gn_defined FPos_q) (pPos_gn_defined FPos_r)) (=
        (fNat*Nat->Nat___+__ (fPos->Nat_gn_inj_Pos_Nat FPos_p) (fNat*Nat->Nat___+__ (fPos
       ->Nat_gn_inj_Pos_Nat FPos_q) (fPos->Nat_gn_inj_Pos_Nat FPos_r))) (fNat*Nat->Nat___
       +__ (fNat*Nat->Nat___+__ (fPos->Nat_gn_inj_Pos_Nat FPos_p) (fPos->
       Nat_gn_inj_Pos_Nat FPos_q)) (fPos->Nat_gn_inj_Pos_Nat FPos_r))))))
51 (prove :axiom add_comm (forall ((FPos_p s_Pos) (FPos_q s_Pos)) (=> (and (
       pPos_gn_defined FPos_p) (pPos_gn_defined FPos_q)) (= (fNat*Nat->Nat___+__ (fPos->
       Nat_gn_inj_Pos_Nat FPos_p) (fPos->Nat_gn_inj_Pos_Nat FPos_q)) (fNat*Nat->Nat___+__
        (fPos->Nat_gn_inj_Pos_Nat FPos_q) (fPos->Nat_gn_inj_Pos_Nat FPos_p))))))
52 (prove :axiom add_suc_right (forall ((FPos_p s_Pos) (FPos_q s_Pos)) (=> (and (
       pPos_gn_defined FPos_p) (pPos_gn_defined FPos_q)) (= (fNat*Nat->Nat___+__ (fPos->
       Nat_gn_inj_Pos_Nat FPos_p) (fPos->Nat_gn_inj_Pos_Nat (fPos->Pos_suc FPos_q))) (
       fNat->Nat_suc (fNat*Nat->Nat___+__ (fPos->Nat_gn_inj_Pos_Nat FPos_p) (fPos->
       Nat_gn_inj_Pos_Nat FPos_q)))))))
```

Listing A.9: TIP translation of `spec Nat` from Listing A.3 using a non-persistently liberal comorphism with features not supported by Zipperposition kept for readability

```
1  (declare-datatypes ((s_Pos 0) (s_Nat 0)) (((fPos_1) (fPos_gn_bottom_Pos) (fPos->
       Pos_suc (i1_fPos->Pos_suc s_Pos))) ((fNat_0) (fNat_gn_bottom_Nat) (fNat->Nat_suc (
       i1_fNat->Nat_suc s_Nat)))))
2  (declare-fun pNat*Nat___<=__ (s_Nat s_Nat) Bool)
3  (declare-fun pNat_gn_defined (s_Nat) Bool)
4  (declare-fun pPos_gn_defined (s_Pos) Bool)
5  (declare-fun fNat*Nat->Nat___+__ (s_Nat s_Nat) s_Nat)
6  (declare-fun fPos->Nat_gn_inj_Pos_Nat (s_Pos) s_Nat)
7  (declare-fun fNat->Pos_gn_proj_Nat_Pos (s_Nat) s_Pos)
8  (declare-fun fNat*Nat->Nat_max (s_Nat s_Nat) s_Nat)
9  (declare-fun fNat*Nat->Nat_min (s_Nat s_Nat) s_Nat)
10 (declare-fun fNat->Nat_pre (s_Nat) s_Nat)
11 (assert :axiom ga_nonEmpty_Nat (exists ((FNat_x s_Nat)) (pNat_gn_defined FNat_x)))
12 (assert :axiom ga_notDefBottom_Nat (not (pNat_gn_defined (fNat_gn_bottom_Nat))))
13 (assert :axiom ga_nonEmpty_Pos (exists ((FPos_x s_Pos)) (pPos_gn_defined FPos_x)))
14 (assert :axiom ga_notDefBottom_Pos (not (pPos_gn_defined (fPos_gn_bottom_Pos))))
15 (assert :axiom ga_strictness_0 (pNat_gn_defined (fNat_0)))
16 (assert :axiom ga_strictness_1 (pPos_gn_defined (fPos_1)))
17 (assert :axiom ga_strictness___+__ (forall ((FNat_x_1 s_Nat) (FNat_x_2 s_Nat)) (= (
       pNat_gn_defined (fNat*Nat->Nat___+__ FNat_x_1 FNat_x_2)) (and (pNat_gn_defined
       FNat_x_1) (pNat_gn_defined FNat_x_2)))))
18 (assert :axiom ga_strictness_gn_inj_Pos_Nat (forall ((FPos_x_1 s_Pos)) (= (
       pNat_gn_defined (fPos->Nat_gn_inj_Pos_Nat FPos_x_1)) (pPos_gn_defined FPos_x_1))))
19 (assert :axiom ga_strictness_gn_proj_Nat_Pos (forall ((FNat_x_1 s_Nat)) (=> (
       pPos_gn_defined (fNat->Pos_gn_proj_Nat_Pos FNat_x_1)) (pNat_gn_defined FNat_x_1)))
       )
20 (assert :axiom ga_strictness_max (forall ((FNat_x_1 s_Nat) (FNat_x_2 s_Nat)) (= (
       pNat_gn_defined (fNat*Nat->Nat_max FNat_x_1 FNat_x_2)) (and (pNat_gn_defined
       FNat_x_1) (pNat_gn_defined FNat_x_2)))))
21 (assert :axiom ga_strictness_min (forall ((FNat_x_1 s_Nat) (FNat_x_2 s_Nat)) (= (
       pNat_gn_defined (fNat*Nat->Nat_min FNat_x_1 FNat_x_2)) (and (pNat_gn_defined
       FNat_x_1) (pNat_gn_defined FNat_x_2)))))
22 (assert :axiom ga_strictness_pre (forall ((FNat_x_1 s_Nat)) (=> (pNat_gn_defined (fNat
       ->Nat_pre FNat_x_1)) (pNat_gn_defined FNat_x_1))))
23 (assert :axiom ga_strictness_suc (forall ((FNat_x_1 s_Nat)) (= (pNat_gn_defined (fNat
       ->Nat_suc FNat_x_1)) (pNat_gn_defined FNat_x_1))))
24 (assert :axiom ga_strictness_suc_1 (forall ((FPos_x_1 s_Pos)) (= (pPos_gn_defined (
       fPos->Pos_suc FPos_x_1)) (pPos_gn_defined FPos_x_1))))
25 (assert :axiom ga_predicate_strictness___<=__ (forall ((FNat_x_1 s_Nat) (FNat_x_2
       s_Nat)) (=> (pNat*Nat___<=__ FNat_x_1 FNat_x_2) (and (pNat_gn_defined FNat_x_1) (
       pNat_gn_defined FNat_x_2)))))
26 (assert :axiom ga_function_monotonicity (forall ((FPos_x1 s_Pos)) (=> (pPos_gn_defined
        FPos_x1) (= (fNat->Nat_suc (fPos->Nat_gn_inj_Pos_Nat FPos_x1)) (fPos->
       Nat_gn_inj_Pos_Nat (fPos->Pos_suc FPos_x1))))))
27 (assert :axiom ga_embedding_injectivity_Pos_to_Nat (forall ((FPos_x s_Pos) (FPos_y
       s_Pos)) (=> (and (pPos_gn_defined FPos_x) (pPos_gn_defined FPos_y)) (=> (and (= (
       fPos->Nat_gn_inj_Pos_Nat FPos_x) (fPos->Nat_gn_inj_Pos_Nat FPos_y)) (
       pNat_gn_defined (fPos->Nat_gn_inj_Pos_Nat FPos_x))) (and (= FPos_x FPos_y) (
       pPos_gn_defined FPos_x))))))
28 (assert :axiom ga_projection_injectivity_Nat_to_Pos (forall ((FNat_x s_Nat) (FNat_y
       s_Nat)) (=> (and (pNat_gn_defined FNat_x) (pNat_gn_defined FNat_y)) (=> (and (= (
       fNat->Pos_gn_proj_Nat_Pos FNat_x) (fNat->Pos_gn_proj_Nat_Pos FNat_y)) (
       pPos_gn_defined (fNat->Pos_gn_proj_Nat_Pos FNat_x))) (and (= FNat_x FNat_y) (
       pNat_gn_defined FNat_x))))))
29 (assert :axiom ga_projection_Nat_to_Pos (forall ((FPos_x s_Pos)) (=> (pPos_gn_defined
       FPos_x) (and (= (fNat->Pos_gn_proj_Nat_Pos (fPos->Nat_gn_inj_Pos_Nat FPos_x))
       FPos_x) (pPos_gn_defined (fNat->Pos_gn_proj_Nat_Pos (fPos->Nat_gn_inj_Pos_Nat
       FPos_x)))))))
30 (assert :axiom ga_injective_suc (forall ((FPos_X1 s_Pos) (FPos_Y1 s_Pos)) (=> (and (
       pPos_gn_defined FPos_X1) (pPos_gn_defined FPos_Y1)) (= (= (fPos->Pos_suc FPos_X1)
       (fPos->Pos_suc FPos_Y1)) (= FPos_X1 FPos_Y1)))))
31 (assert :axiom ga_disjoint_1_suc (forall ((FPos_Y1 s_Pos)) (=> (pPos_gn_defined
       FPos_Y1) (not (= (fPos_1) (fPos->Pos_suc FPos_Y1))))))
32 (assert :axiom ga_generated_Pos (forall ((PPos_gn_P_Pos (=> s_Pos Bool))) (=> (and (@
       PPos_gn_P_Pos (fPos_1)) (@ PPos_gn_P_Pos (fPos_gn_bottom_Pos)) (forall ((FPos_y_1
       s_Pos)) (=> (@ PPos_gn_P_Pos FPos_y_1) (@ PPos_gn_P_Pos (fPos->Pos_suc FPos_y_1)))
       )) (forall ((FPos_x_1 s_Pos)) (@ PPos_gn_P_Pos FPos_x_1)))))
33 (assert :axiom ga_selector_pre (forall ((FNat_X1 s_Nat)) (=> (pNat_gn_defined FNat_X1)
        (= (fNat->Nat_pre (fNat->Nat_suc FNat_X1)) FNat_X1))))
```

```
34 (assert :axiom ga_injective_suc_1 (forall ((FNat_X1 s_Nat) (FNat_Y1 s_Nat)) (=> (and (
       pNat_gn_defined FNat_X1) (pNat_gn_defined FNat_Y1)) (= (= (fNat->Nat_suc FNat_X1)
       (fNat->Nat_suc FNat_Y1)) (= FNat_X1 FNat_Y1)))))
35 (assert :axiom ga_disjoint_0_suc (forall ((FNat_Y1 s_Nat)) (=> (pNat_gn_defined
       FNat_Y1) (not (= (fNat_0) (fNat->Nat_suc FNat_Y1))))))
36 (assert :axiom ga_selector_undef_pre_0 (not (pNat_gn_defined (fNat->Nat_pre (fNat_0)))
       ))
37 (assert :axiom ga_generated_Nat (forall ((PNat_gn_P_Nat (=> s_Nat Bool))) (=> (and (@
       PNat_gn_P_Nat (fNat_0)) (@ PNat_gn_P_Nat (fNat_gn_bottom_Nat)) (forall ((FNat_y_1
       s_Nat)) (=> (@ PNat_gn_P_Nat FNat_y_1) (@ PNat_gn_P_Nat (fNat->Nat_suc FNat_y_1)))
       )) (forall ((FNat_x_1 s_Nat)) (@ PNat_gn_P_Nat FNat_x_1)))))
38 (assert :axiom Pos_def (= (fNat->Nat_suc (fNat_0)) (fPos->Nat_gn_inj_Pos_Nat (fPos_1))
       ))
39 (assert :axiom add_0 (forall ((FNat_m s_Nat)) (=> (pNat_gn_defined FNat_m) (= (fNat*
       Nat->Nat___+__ (fNat_0) FNat_m) FNat_m))))
40 (assert :axiom add_suc (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (fNat*Nat->Nat___+__ (fNat->
       Nat_suc FNat_n) FNat_m) (fNat->Nat_suc (fNat*Nat->Nat___+__ FNat_n FNat_m))))))
41 (assert :axiom leq_def1 (forall ((FNat_n s_Nat)) (=> (pNat_gn_defined FNat_n) (pNat*
       Nat___<=__ (fNat_0) FNat_n))))
42 (assert :axiom leq_def2 (forall ((FNat_n s_Nat)) (=> (pNat_gn_defined FNat_n) (not (
       pNat*Nat___<=__ (fNat->Nat_suc FNat_n) (fNat_0))))))
43 (assert :axiom leq_def3 (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (pNat*Nat___<=__ (fNat->
       Nat_suc FNat_m) (fNat->Nat_suc FNat_n)) (pNat*Nat___<=__ FNat_m FNat_n)))))
44 (assert :axiom min_def (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (fNat*Nat->Nat_min FNat_m
       FNat_n) (ite (pNat*Nat___<=__ FNat_m FNat_n) FNat_m FNat_n)))))
45 (assert :axiom max_def (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (fNat*Nat->Nat_max FNat_m
       FNat_n) (ite (pNat*Nat___<=__ FNat_m FNat_n) FNat_n FNat_m)))))
46 (prove :axiom add_0_right (forall ((FPos_p s_Pos)) (=> (pPos_gn_defined FPos_p) (= (
       fNat*Nat->Nat___+__ (fPos->Nat_gn_inj_Pos_Nat FPos_p) (fNat_0)) (fPos->
       Nat_gn_inj_Pos_Nat FPos_p)))))
47 (prove :axiom add_assoc (forall ((FPos_p s_Pos) (FPos_q s_Pos) (FPos_r s_Pos)) (=> (
       and (pPos_gn_defined FPos_p) (pPos_gn_defined FPos_q) (pPos_gn_defined FPos_r)) (=
        (fNat*Nat->Nat___+__ (fPos->Nat_gn_inj_Pos_Nat FPos_p) (fNat*Nat->Nat___+__ (fPos
       ->Nat_gn_inj_Pos_Nat FPos_q) (fPos->Nat_gn_inj_Pos_Nat FPos_r))) (fNat*Nat->Nat___
       +__ (fNat*Nat->Nat___+__ (fPos->Nat_gn_inj_Pos_Nat FPos_p) (fPos->
       Nat_gn_inj_Pos_Nat FPos_q)) (fPos->Nat_gn_inj_Pos_Nat FPos_r))))))
48 (prove :axiom add_comm (forall ((FPos_p s_Pos) (FPos_q s_Pos)) (=> (and (
       pPos_gn_defined FPos_p) (pPos_gn_defined FPos_q)) (= (fNat*Nat->Nat___+__ (fPos->
       Nat_gn_inj_Pos_Nat FPos_p) (fPos->Nat_gn_inj_Pos_Nat FPos_q)) (fNat*Nat->Nat___+__
        (fPos->Nat_gn_inj_Pos_Nat FPos_q) (fPos->Nat_gn_inj_Pos_Nat FPos_p))))))
49 (prove :axiom add_suc_right (forall ((FPos_p s_Pos) (FPos_q s_Pos)) (=> (and (
       pPos_gn_defined FPos_p) (pPos_gn_defined FPos_q)) (= (fNat*Nat->Nat___+__ (fPos->
       Nat_gn_inj_Pos_Nat FPos_p) (fPos->Nat_gn_inj_Pos_Nat (fPos->Pos_suc FPos_q))) (
       fNat->Nat_suc (fNat*Nat->Nat___+__ (fPos->Nat_gn_inj_Pos_Nat FPos_p) (fPos->
       Nat_gn_inj_Pos_Nat FPos_q)))))))
```

Listing A.10: TIP translation of `spec Nat` from Listing A.4 using a persistently liberal comorphism with features not supported by Zipperposition kept for readability

```
(declare-sort s_Nat 0)
(declare-sort s_Pos 0)
(declare-fun pNat*Nat___<=__ (s_Nat s_Nat) Bool)
(declare-fun pNat_gn_defined (s_Nat) Bool)
(declare-fun pPos_gn_defined (s_Pos) Bool)
(declare-const fNat_0 s_Nat)
(declare-const fPos_1 s_Pos)
(declare-fun fNat*Nat->Nat___+__ (s_Nat s_Nat) s_Nat)
(declare-const fNat_gn_bottom_Nat s_Nat)
(declare-const fPos_gn_bottom_Pos s_Pos)
(declare-fun fPos->Nat_gn_inj_Pos_Nat (s_Pos) s_Nat)
(declare-fun fNat->Pos_gn_proj_Nat_Pos (s_Nat) s_Pos)
(declare-fun fNat*Nat->Nat_max (s_Nat s_Nat) s_Nat)
(declare-fun fNat*Nat->Nat_min (s_Nat s_Nat) s_Nat)
(declare-fun fNat->Nat_pre (s_Nat) s_Nat)
(declare-fun fNat->Nat_suc (s_Nat) s_Nat)
(declare-fun fPos->Pos_suc (s_Pos) s_Pos)
(assert :axiom ga_nonEmpty_Nat (exists ((FNat_x s_Nat)) (pNat_gn_defined FNat_x)))
(assert :axiom ga_notDefBottom_Nat (forall ((FNat_x s_Nat)) (= (not (pNat_gn_defined
    FNat_x)) (= FNat_x (fNat_gn_bottom_Nat)))))
(assert :axiom ga_nonEmpty_Pos (exists ((FPos_x s_Pos)) (pPos_gn_defined FPos_x)))
(assert :axiom ga_notDefBottom_Pos (forall ((FPos_x s_Pos)) (= (not (pPos_gn_defined
    FPos_x)) (= FPos_x (fPos_gn_bottom_Pos)))))
(assert :axiom ga_strictness_0 (pNat_gn_defined (fNat_0)))
(assert :axiom ga_strictness_1 (pPos_gn_defined (fPos_1)))
(assert :axiom ga_strictness___+__ (forall ((FNat_x_1 s_Nat) (FNat_x_2 s_Nat)) (= (
    pNat_gn_defined (fNat*Nat->Nat___+__ FNat_x_1 FNat_x_2)) (and (pNat_gn_defined
    FNat_x_1) (pNat_gn_defined FNat_x_2)))))
(assert :axiom ga_strictness_gn_inj_Pos_Nat (forall ((FPos_x_1 s_Pos)) (= (
    pNat_gn_defined (fPos->Nat_gn_inj_Pos_Nat FPos_x_1)) (pPos_gn_defined FPos_x_1))))
(assert :axiom ga_strictness_gn_proj_Nat_Pos (forall ((FNat_x_1 s_Nat)) (=> (
    pPos_gn_defined (fNat->Pos_gn_proj_Nat_Pos FNat_x_1)) (pNat_gn_defined FNat_x_1)))
    )
(assert :axiom ga_strictness_max (forall ((FNat_x_1 s_Nat) (FNat_x_2 s_Nat)) (= (
    pNat_gn_defined (fNat*Nat->Nat_max FNat_x_1 FNat_x_2)) (and (pNat_gn_defined
    FNat_x_1) (pNat_gn_defined FNat_x_2)))))
(assert :axiom ga_strictness_min (forall ((FNat_x_1 s_Nat) (FNat_x_2 s_Nat)) (= (
    pNat_gn_defined (fNat*Nat->Nat_min FNat_x_1 FNat_x_2)) (and (pNat_gn_defined
    FNat_x_1) (pNat_gn_defined FNat_x_2)))))
(assert :axiom ga_strictness_pre (forall ((FNat_x_1 s_Nat)) (=> (pNat_gn_defined (fNat
    ->Nat_pre FNat_x_1)) (pNat_gn_defined FNat_x_1))))
(assert :axiom ga_strictness_suc (forall ((FNat_x_1 s_Nat)) (= (pNat_gn_defined (fNat
    ->Nat_suc FNat_x_1)) (pNat_gn_defined FNat_x_1))))
(assert :axiom ga_strictness_suc_1 (forall ((FPos_x_1 s_Pos)) (= (pPos_gn_defined (
    fPos->Pos_suc FPos_x_1)) (pPos_gn_defined FPos_x_1))))
(assert :axiom ga_predicate_strictness___<=__ (forall ((FNat_x_1 s_Nat) (FNat_x_2
    s_Nat)) (=> (pNat*Nat___<=__ FNat_x_1 FNat_x_2) (and (pNat_gn_defined FNat_x_1) (
    pNat_gn_defined FNat_x_2)))))
(assert :axiom ga_function_monotonicity (forall ((FPos_x1 s_Pos)) (=> (pPos_gn_defined
     FPos_x1) (= (fNat->Nat_suc (fPos->Nat_gn_inj_Pos_Nat FPos_x1)) (fPos->
    Nat_gn_inj_Pos_Nat (fPos->Pos_suc FPos_x1))))))
(assert :axiom ga_embedding_injectivity_Pos_to_Nat (forall ((FPos_x s_Pos) (FPos_y
    s_Pos)) (=> (and (pPos_gn_defined FPos_x) (pPos_gn_defined FPos_y)) (=> (and (= (
    fPos->Nat_gn_inj_Pos_Nat FPos_x) (fPos->Nat_gn_inj_Pos_Nat FPos_y)) (
    pNat_gn_defined (fPos->Nat_gn_inj_Pos_Nat FPos_x))) (and (= FPos_x FPos_y) (
    pPos_gn_defined FPos_x))))))
(assert :axiom ga_projection_injectivity_Nat_to_Pos (forall ((FNat_x s_Nat) (FNat_y
    s_Nat)) (=> (and (pNat_gn_defined FNat_x) (pNat_gn_defined FNat_y)) (=> (and (= (
    fNat->Pos_gn_proj_Nat_Pos FNat_x) (fNat->Pos_gn_proj_Nat_Pos FNat_y)) (
    pPos_gn_defined (fNat->Pos_gn_proj_Nat_Pos FNat_x))) (and (= FNat_x FNat_y) (
    pNat_gn_defined FNat_x))))))
(assert :axiom ga_projection_Nat_to_Pos (forall ((FPos_x s_Pos)) (=> (pPos_gn_defined
    FPos_x) (and (= (fNat->Pos_gn_proj_Nat_Pos (fPos->Nat_gn_inj_Pos_Nat FPos_x))
    FPos_x) (pPos_gn_defined (fNat->Pos_gn_proj_Nat_Pos (fPos->Nat_gn_inj_Pos_Nat
    FPos_x)))))))
(assert :axiom ga_injective_suc (forall ((FPos_X1 s_Pos) (FPos_Y1 s_Pos)) (=> (and (
    pPos_gn_defined FPos_X1) (pPos_gn_defined FPos_Y1)) (= (= (fPos->Pos_suc FPos_X1)
    (fPos->Pos_suc FPos_Y1)) (= FPos_X1 FPos_Y1)))))
(assert :axiom ga_disjoint_1_suc (forall ((FPos_Y1 s_Pos)) (=> (pPos_gn_defined
    FPos_Y1) (not (= (fPos_1) (fPos->Pos_suc FPos_Y1))))))
```

```
39 (assert :axiom ga_generated_Pos (forall ((PPos_gn_P_Pos (=> s_Pos Bool))) (=> (and (@
       PPos_gn_P_Pos (fPos_1)) (@ PPos_gn_P_Pos (fPos_gn_bottom_Pos)) (forall ((FPos_y_1
       s_Pos)) (=> (@ PPos_gn_P_Pos FPos_y_1) (@ PPos_gn_P_Pos (fPos->Pos_suc FPos_y_1)))
       )) (forall ((FPos_x_1 s_Pos)) (@ PPos_gn_P_Pos FPos_x_1)))))
40 (assert :axiom ga_selector_pre (forall ((FNat_X1 s_Nat)) (=> (pNat_gn_defined FNat_X1)
        (= (fNat->Nat_pre (fNat->Nat_suc FNat_X1)) FNat_X1))))
41 (assert :axiom ga_injective_suc_1 (forall ((FNat_X1 s_Nat) (FNat_Y1 s_Nat)) (=> (and (
       pNat_gn_defined FNat_X1) (pNat_gn_defined FNat_Y1)) (= (= (fNat->Nat_suc FNat_X1)
       (fNat->Nat_suc FNat_Y1)) (= FNat_X1 FNat_Y1)))))
42 (assert :axiom ga_disjoint_0_suc (forall ((FNat_Y1 s_Nat)) (=> (pNat_gn_defined
       FNat_Y1) (not (= (fNat_0) (fNat->Nat_suc FNat_Y1))))))
43 (assert :axiom ga_selector_undef_pre_0 (not (pNat_gn_defined (fNat->Nat_pre (fNat_0)))
       ))
44 (assert :axiom ga_generated_Nat (forall ((PNat_gn_P_Nat (=> s_Nat Bool))) (=> (and (@
       PNat_gn_P_Nat (fNat_0)) (@ PNat_gn_P_Nat (fNat_gn_bottom_Nat)) (forall ((FNat_y_1
       s_Nat)) (=> (@ PNat_gn_P_Nat FNat_y_1) (@ PNat_gn_P_Nat (fNat->Nat_suc FNat_y_1)))
       )) (forall ((FNat_x_1 s_Nat)) (@ PNat_gn_P_Nat FNat_x_1)))))
45 (assert :axiom Pos_def (= (fNat->Nat_suc (fNat_0)) (fPos->Nat_gn_inj_Pos_Nat (fPos_1))
       ))
46 (assert :axiom add_0 (forall ((FNat_m s_Nat)) (=> (pNat_gn_defined FNat_m) (= (fNat*
       Nat->Nat___+__ (fNat_0) FNat_m) FNat_m))))
47 (assert :axiom add_suc (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (fNat*Nat->Nat___+__ (fNat->
       Nat_suc FNat_n) FNat_m) (fNat->Nat_suc (fNat*Nat->Nat___+__ FNat_n FNat_m))))))
48 (assert :axiom leq_def1 (forall ((FNat_n s_Nat)) (=> (pNat_gn_defined FNat_n) (pNat*
       Nat___<=__ (fNat_0) FNat_n))))
49 (assert :axiom leq_def2 (forall ((FNat_n s_Nat)) (=> (pNat_gn_defined FNat_n) (not (
       pNat*Nat___<=__ (fNat->Nat_suc FNat_n) (fNat_0))))))
50 (assert :axiom leq_def3 (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (pNat*Nat___<=__ (fNat->
       Nat_suc FNat_m) (fNat->Nat_suc FNat_n)) (pNat*Nat___<=__ FNat_m FNat_n)))))
51 (assert :axiom min_def (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (fNat*Nat->Nat_min FNat_m
       FNat_n) (ite (pNat*Nat___<=__ FNat_m FNat_n) FNat_m FNat_n)))))
52 (assert :axiom max_def (forall ((FNat_m s_Nat) (FNat_n s_Nat)) (=> (and (
       pNat_gn_defined FNat_m) (pNat_gn_defined FNat_n)) (= (fNat*Nat->Nat_max FNat_m
       FNat_n) (ite (pNat*Nat___<=__ FNat_m FNat_n) FNat_n FNat_m)))))
53 (prove :axiom add_0_right (forall ((FPos_p s_Pos)) (=> (pPos_gn_defined FPos_p) (= (
       fNat*Nat->Nat___+__ (fPos->Nat_gn_inj_Pos_Nat FPos_p) (fNat_0)) (fPos->
       Nat_gn_inj_Pos_Nat FPos_p)))))
54 (prove :axiom add_assoc (forall ((FPos_p s_Pos) (FPos_q s_Pos) (FPos_r s_Pos)) (=> (
       and (pPos_gn_defined FPos_p) (pPos_gn_defined FPos_q) (pPos_gn_defined FPos_r)) (=
        (fNat*Nat->Nat___+__ (fPos->Nat_gn_inj_Pos_Nat FPos_p) (fNat*Nat->Nat___+__ (fPos
       ->Nat_gn_inj_Pos_Nat FPos_q) (fPos->Nat_gn_inj_Pos_Nat FPos_r))) (fNat*Nat->Nat___
       +__ (fNat*Nat->Nat___+__ (fPos->Nat_gn_inj_Pos_Nat FPos_p) (fPos->
       Nat_gn_inj_Pos_Nat FPos_q)) (fPos->Nat_gn_inj_Pos_Nat FPos_r))))))
55 (prove :axiom add_comm (forall ((FPos_p s_Pos) (FPos_q s_Pos)) (=> (and (
       pPos_gn_defined FPos_p) (pPos_gn_defined FPos_q)) (= (fNat*Nat->Nat___+__ (fPos->
       Nat_gn_inj_Pos_Nat FPos_p) (fPos->Nat_gn_inj_Pos_Nat FPos_q)) (fNat*Nat->Nat___+__
        (fPos->Nat_gn_inj_Pos_Nat FPos_q) (fPos->Nat_gn_inj_Pos_Nat FPos_p))))))
56 (prove :axiom add_suc_right (forall ((FPos_p s_Pos) (FPos_q s_Pos)) (=> (and (
       pPos_gn_defined FPos_p) (pPos_gn_defined FPos_q)) (= (fNat*Nat->Nat___+__ (fPos->
       Nat_gn_inj_Pos_Nat FPos_p) (fPos->Nat_gn_inj_Pos_Nat (fPos->Pos_suc FPos_q))) (
       fNat->Nat_suc (fNat*Nat->Nat___+__ (fPos->Nat_gn_inj_Pos_Nat FPos_p) (fPos->
       Nat_gn_inj_Pos_Nat FPos_q)))))))
```

Listing A.11: TIP translation of `spec Nat` from Listing A.4 using a non-persistently liberal comorphism with features not supported by Zipperposition kept for readability

I herewith assure that I wrote the present thesis titled *Induction Provers in Hets: Leveraging the Tons of Inductive Problems language and tools to talk to more Automated Theorem Provers* independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or to their meaning.

I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a penalty by the law enforcement agency.

Potsdam, November 25, 2022

(Tom Kranz)