Combining Provers in the *Theorema* System

Elena Tomuta, Bruno Buchberger
Research Institute for Symbolic Computation
A-4232 Schloss Hagenberg, Austria
{FirstName.LastName}@RISC.uni-linz.ac.at

January 30, 1998

Abstract

The *Theorema* project aims at extending current computer algebra systems by facilities for supporting mathematical proving. This paper reports on a mechanism for combining provers currently under development in the frame of the project. We build on the traditional concepts of proof-planning but adapt them to the specific setting of *Theorema* : existance of several autonomous special provers with built-in heuristics controlled by a master natural deduction style prover. The system is implemented in *Mathematica* whose functional language features enable the implementation of proof search algorithms in a very natural way.

1 Introduction

The *Theorema* project aims at extending current computer algebra systems by facilities for supporting proving. In the development the concept of *functor* is a crucial one because it allows the unification of algorithmic and non-algorithmic mathematics (see [Buchberger 1996]). Building up mathematics by functors has the natural consequence that provers for various areas of mathematics can be structured in a similar way. The idea is thus to associate special provers with functors such that the knowledge about algebraic structure which is contained in functors can be used for reducing the complexity of proof-search. See also [Buchberger 1993] for a discussion on the gap between computer algebra systems and mathematical assistants.

In practice the special provers corresponding to various functors will need to support various kinds of reasoning steps like natural deduction, handling of equality, case distinctions, applying definitions, induction and computation. Therefore a mechanism for *combining provers* is of particular importance for the development of the system.

The *Theorema* Mathematical Assistant program is based on a natural deduction style predicate logic prover ([Buchberger, Jebelean 1997]). Our aim is to develop a framework which enables us to quickly incorporate special proof steps as the ones listed above into the proofs performed by the natural deduction prover. On the other hand we want to keep the various proof engines separate and enable users to quickly recombine existing proof modules and add their own. Thus we need a way to specify how various proof engines interact with the natural deduction prover and how they interact with each other in the process of searching for a proof. It should also be possible to formulate, by means of a

Presented at the Sixth Rhine Workshop on Computer Algebra, Sankt Augustin, Germany April, 1998.

Supported by the Austrian Forschungsfoerderungsfonds (FWF), SFB project F1302.

simple language, constraints to be used for reducing the proof search space of individual proofs or classes of proofs.

Based on the concepts of plan operators ([Fikes, Nilsson, 1970]), and tactics ([Gordon et al. 1979]), several automated theorem proving systems have taken approaches similar to the one presented in this paper. Bundy ([Bundy, 1988]) introduced the notion of method by extending a tactic with its specification which can be used by a planner; the Oyster-Clam system ([Bundy et al., 1990]) implements these concepts. [Huang et al., 1994] proposes a different notion of a method, in which the list of logical inference rules applied by the tactic are declared as part of the method definition. The procedural part of the tactic is then generated automatically by means of an interpreter from the declarative part. In this way all knowledge about the logical operations performed by the method is contained in the declarative part making it easier to automatically modify the method.

We follow the same approach in that special provers are operators (or methods) which act on a global state – an assignment of *Mathematica* terms to variables. Special provers apply to certain *proof situations* (pairs consisting in a goal and a knowledge base) and return a derivation tree. Derivation trees record the steps performed by the respective prover (for use in generating a natural language representation of the whole proof) but also enable the special prover to exercise control over the way the proof proceeds. The difference from the approaches we have seen so far consists in the fact that we see special provers as encapsulating both proof control heuristics and procedural content. This view follows from the aim of *Theorema* to unify in one system specialised proof engines based on algebraic techniques, like the Knuth-Bendix procedure for proving in equational logic, Collins' quantifier elimination method or the Groebner Bases and characteristic set methods used in proving some classes of theorems in elementary geometry ([Buchberger 1993]). Clearly, an existing prover incorporated into *Theorema* will be wrapped in a software module performing the necessary translations between the data staructures of the respective prover and the ones of *Theorema* .

At each step the proof system is characterised by a state. The variables of the system state are used in proof control and some have special meaning for the proof search engine. Others can be used by the provers to communicate heuristic information. The model is thus similar to the blackboard paradigm ([Hewitt, 1977]) where the proof object and system state form the blackboard and the special provers are actors.

The interaction of the provers participating in the proof can be controlled by means of a simple language. The intention is to give the programmer of new, domain customized proof engines maximum flexibility and ease in specifying that a proof should be carried out according to certain constraints.

The paper is structured as follows: section 2 briefly explains the notion of special prover in the *Theorema* system. In section 3 we define the notion of derivation trees and show how they are interpreted by the proof search engine. The system state and the possible modes of interacting with it are presented in section 4. Section 5 defines proof control constraints. Section 6 presents the proof search algorithm. We conclude with an example in section 7 and some suggestions for further research in section 8. The Appendix shows a proof generated by the system.

2 Special Provers

Special provers are *Mathematica* programs taking as argument a proof situation (pair consisting in a goal and a knowledge base) and returning a derivation tree or *Null*, if the special prover fails on the given proof situation. Taking advantage of the pattern matching features of *Mathematica* in the implementation of special provers it is easy to limit the applicability of a special prover to proof situations of a certain form. Constraints on the subterms of the proof situations passed as argument can be defined as *Mathematica* predicates.

3 Derivation Trees

The proof search engine we describe has no notion of forward or backward reasoning. It just sees a partial *derivation tree* possibly containing several unsolved proof situations and it must decide which of the proof situations it should tackle first and what the consequence of solving one proof situation is for the global proof. A derivation tree is a tree with nodes composed of:

- a list of parameters recording the modifications on the proof situation effected by the special prover;
- a set of proof situations to which the initial one is reduced and an indication as to how proving the resulting proof situations relates to proving the initial one.

For instance, when doing a proof by induction, we normally have to prove one or several base cases and the induction step, hence the initial proof situation is proved if and only if we manage to prove each of the resulting ones. In forward proof, when using an equality to rewrite the goal, we reduce an initial proof situation to an equivalent one. The initial proof situation is proved if and only if the resulting one is.
We now give the syntax of derivation trees in Backus-Naur form. A *derivation tree DT* is a *Mathematica* expression generated by the following grammar:

DT	:=	**TProofStep**[*stepParams*, *DTList*] |
		TProofSit[*goal*, *kb*] |
		“proved” |**“disproved”** | **“failed”** | **IF**[DT, DT, DT]
DTList	:=	*void* | *DT* | *DT*, *DTList*
goal	:=	a labeled formula in *Theorema* syntax
kb	:=	a list of labeled formulae in *Theorema* syntax
stepParams	:=	a list of *Mathematica* expressions

The proof engine combines the various separate derivation trees into the global proof tree. We can then view the process of proving as a process of assigning to a proof object one of the (proof) values ”proved”, ”disproved” or ”failed”, having the intuitive meaning of ”the given conjecture is proved”, ”the given conjecture is disproved”, and respectively, ”the system has failed to prove the given conjecture”. Such proof values are assigned bottom up, based on the recursive structure of the derivation tree, so in order to specify the operational semantics of the proof search engine we give rules for annotating each derivation tree with a value of "proved", "disproved" or "failed".

$$BASE1: \frac{}{\textbf{proved}\colon \textit{proved}} \quad BASE2: \frac{}{\textbf{failed}\colon \textit{failed}} \quad BASE3: \frac{}{\textbf{disproved}\colon \textit{disproved}}$$

$$AND1: \frac{DT\colon \textit{disproved}}{\textbf{TProofStep}[DTList,\ DT,\ DTList]\colon \textit{disproved}}$$

$$AND2: \frac{DT\colon \textit{failed}}{\textbf{TProofStep}[DTList,\ DT,\ DTList]\colon \textit{failed}}$$

$$AND3: \frac{DT\colon \textit{proved},\ \textbf{TProofStep}[DTList_1,\ DTList_2\ ]\colon \textit{val}}{\textbf{TProofStep}[DTList_1,\ DT,\ DTList_2\ ]\colon \textit{val}}$$

where $val \in \{proved, disproved, failed\}$.

$$IF1: \frac{DT_1\colon \textit{proved},\ \textbf{TProofStep}[DTList_1,\ DT_2,\ DTList_2]\colon \textit{val}}{\textbf{TProofStep}[DTList_1,\ \textbf{IF}[DT_1,\ DT_2,\ DT_3],\ DTList_2]\colon \textit{val}}$$

where $val \in \{proved, disproved, failed\}$.

$$IF2: \frac{DT_1\colon \textit{disproved},\ \textbf{TProofStep}[DTList_1,\ DT_3,\ DTList_2]\colon \textit{val}}{\textbf{TProofStep}[DTList_1,\ \textbf{IF}[DT_1,\ DT_2,\ DT_3],\ DTList_2]\colon \textit{val}}$$

where $val \in \{proved, disproved, failed\}$.

$$IF3: \frac{DT_1\colon \textit{failed}}{\textbf{TProofStep}[DTList_1,\ \textbf{IF}[DT_1,\ DT_2,\ DT_3],\ DTList_2]\colon \textit{failed}}$$

4 Proof Context

During proof construction it often happens that a prover fires on several proof situations consecutively. Also, facts gathered during proof situation analysis by one prover can influence the heuristic of another prover participating in the proof. For these reasons we provide an abstract data structure called *proof context*, which can be used for communicating data between provers. The proof context allows the user to store and modify an object associated with a key and to retrieve a value associated with a key. From the point of view of lifetime of a proof context variable, the following modes of storing an object are provided:

Persistent: When an object is stored as persistent its value stays the same for the rest of the current proof construction process. Other provers can modify the value of the object.

Dynamic: An object defined as dynamic during evaluation of some proof situation P of the current proof object retains its value only during construction of the subtree of the proof tree originating from node P. Dynamic proof context variables can be implemented very easily via the dynamic scoping construct provided by *Mathematica* ([Wolfram, 1996]).

The proof context can be modified through the following functions:

AddProofContextElem[*key*, *object*, *mode*] Associates the object *object* with the key *key* in the mode specified by *mode*, where *mode* is one of ”persistent” or ”local”.

GetProofContextElem[*key*, *mode*] Retrieves the object associated with *key* in the mode *mode*.

We list below the keys in the proof context data structure that have a built-in meaning associated.

The key **ProversHistory** has associated with it the list of provers which have been called on any of the parent nodes of the current proof situation.
The key **SearchDepth** holds a *Mathematica* interger, giving the maximum depth of the search tree in which a proof is searched. If some branch of the proof search tree increases over the given limit then the respective subproof stops with ”failed”.
The key **ProofSitOrder** holds a *Mathematica* predicate which takes as argument 2 proof situations and returns $True$ if the first one is greater than the second one and $False$ otherwise. This predicate is used for specifying the order in which proof situations are expanded.

5 Proof Control Constraints

In section 2 we saw that it is possible to attach to a special prover a set of *constraints* inherent to its logic and hence fixed for all proofs. In addition to these we allow the specification of constraints which can be defined for individual (sets of) proofs and take into account the proof context. These sets of constraints are *Mathematica* predicates evaluating to $True$ or $False$ and are defined in terms of values in the proof context. Such constraints can be specified by means of a simple language given below. We first introduce the basic concepts of *proof control constraint* and *proof outline*. A proof control constraint controls when a special prover is applied. A proof outline indicates to the proof search engine how to apply special provers during a proof. We now define these concepts precisely.
Definition 1. A *Proof Control Constraint* is a *Mathematica* function which takes as arguments a goal, a knowledge base, and the current proof context and returns $True$ or $False$.
Definition 2. A *Proof Outline* is a list of proof constraint specifications.
Definition 3. A *Proof Constraint Specification* is an expression PCS generated by the following grammar:

$$
\begin{array}{lcl}
PCS & := & \mathrm{ON} < constraint > < DoRepeatClause > \\
 & & < ThenClause > \\
< DoRepeatClause > & := & \mathrm{DO} < SpecialProver > \mid \\
 & & \mathrm{REPEAT} < SpecialProver > \\
< ThenClause > & := & \mathrm{DO} < SpecialProver > \mid \\
 & & \mathrm{THEN} < SpecialProver > \\
< constraint > & := & \text{a proof control constraint} \\
< specialProver > & := & \text{a special prover}
\end{array}
$$

The command **SetProofOutline**, if given as argument a list of comma-separated proof constraint specifications, sets the proof outline to be used by the proof engine to the current one.
The syntax of the command is:
SetProofOutline[$< ProofOutline >$]
where $< ProofOutline >$ is an object complying to Definition 2.
The proof outline is parsed and code is generated which is then plugged into the natural deduction prover to adapt its behaviour to the domain we deal with.
The connectives THEN and REPEAT are used with their the usual "tactical" meaning. Given the special provers SP_1 and SP_2, and the proof situation P, $SP_1\ \ THEN\ \ SP_2$ is a special prover which applies SP_1 to P, then applies SP_2 to all proof situations in the derivation tree $dt = SP_1(P)$.
Given the special prover SP, $REPEAT\ \ SP$ is a special prover which applies SP recursively to a given proof situation and to all open proof situations in the resulting derivation tree until it fails.

6 The Search Algorithm

The *Theorema* system is based on a predicate logic prover in natural deduction style, which, given a proof situation generates a proof object recording the inference rules applied in trying to prove the theorem (see [Buchberger, Jebelean 1997]). The current proof search engine extends the natural deduction master prover by inserting special proof steps into the proof produced by the (master) predicate logic prover. After each application of an inference rule by the predicate logic prover the proof search engine is given a chance to interfere by calling one of the special provers to handle the current goal. The resulting derivation tree is then evaluated in conformance with the constraints defined for the current proof and eventually the predicate logic prover is called again on some of the open proof situations of the derivation tree.
Consequently, the proof search engine is based on 2 mutually recursive functions:

HandleProofSituation handles a single open proof situation resulting after the application of an inference rule by the natural deduction prover.

ExecuteDerivationTree handles the derivation tree generated after one or more special provers have been called.

We sketch the implementations of the 2 functions below.

ExecuteDerivationTree

In:	*dt*	a derivation tree
Out:	*po*	a proof object (in the format of the predicate logic prover)

Steps:

1. Build the list L of open proof situations in *dt* and order it by using the proof context function **OrderProofSit**
2. If L is not empty take P as the first element of L.
3. *po* := HandleProofSituation(P)

HandleProofSituation

In:	*goal*	a formula to be proven
	kb	a list of assumptions (knowledge base)
Out:	*po*	a derivation tree

Steps:

1. Build a list L of applicable provers whose proof control constraints are satisfied by $\{goal, kb\}$
2. If the list is empty
   po := PND(*goal*, *kb*);
   return *po*;
   otherwise
   goto 3.
3. Select one prover P of the list L and remove P from L.
4. Set the derivation tree $dt := P(\ goal,\ kb)$
5. If dt = Null
   if L is empty goto 2 else goto 3
5. *po* := ExecuteDerivationTree(*dt*);
6. If *po* has proof value **"failed"** goto 2.
7. return *po*;

Remarks

a) In the algorithm above backtracking occurs at step 6 in HandleProofSituation : if calling a prover results in a failed proof object then we discard the respective subtree and try to apply the next prover in the list of applicable ones.

b) The 2 functions presented above only control the parts of the proof which are not performed by the predicate logic prover. The predicate logic prover incorporates heuristics, in that it decides e.g. which of the applicable inference rules to apply and the order in which to prove the proof situations open at one point. The predicate logic prover repeatedly "challenges" the proof search engine during a proof with open proof situations which some special prover may be able to handle or not.

7 An Example

We implemented a mechanism allowing the user to define functors and domains and prove properties involving functors. From the definition of a functor input by the user a set of special proof modules are generated automatically. These proof modules basically handle type constraints, apply definitions and perform knowledge base transformations meant to reduce the complexity of subproofs. For details on the functor language see [Buchberger et al., 1997]. In this section we only give an example of a proof automatically generated by the system. A new, more sophisticated functor language is currently being implemented in the *Theorema* system. The toy language we use in this example fits its purpose of allowing us to experiment with the way functor-specific knowledge can be used to control proofs.
Let A be a ring with respect to the two binary operations $\oplus$ and $\otimes$ and let C be the cartesian product of A with itself. We denote by $\ominus$ be the inverse operation of $\oplus$ in A. We define the operations $\oplus$ and $\otimes$ over C, in terms of the operations in A as follows:

$$\begin{array}{rl} ADD. & (r_1, i_1) \oplus (r_2, i_2) = (r_1 \oplus r_2, i_1 \oplus i_2), \forall (r_1, i_1), (r_2, i_2) \in C \\ MULT. & (r_1, i_1) \otimes (r_2, i_2) = (r_1 \otimes r_2 \ominus (i_1 \otimes i_2), (r_1 \otimes i_2) \oplus (i_1 \otimes r_2)), \forall (r_1, i_1), (r_2, i_2) \in C \end{array}$$

The construction of C from A is the well known one used in building the complex numbers over the reals and the following easy theorem holds:
Theorem If A is a ring then C is a ring.
In *Theorema* we can formalize this by defining the functor **Complexification** defining the operators and carrier set of C in terms of the corresponding notions in A. We can formulate the property that A is an integral domain and C a ring and prove the theorem given above. The proof is performed automatically by the system and involves proof steps performed by the natural deduction prover and the functor prover. For brevity we prove here only one of the ring axioms, namely the fact that C is closed with respect to addition. We note however, that the complete theorem was proved by the system fully automatically. The needed concepts are defined in *Theorema* as follows:

7.1 The Functor Complexification

$$\text{Complexification}[\boldsymbol{A}] \equiv$$
$$\text{Functor}[\langle \boldsymbol{C}, \text{"}\boldsymbol{\epsilon}\text{"}, \text{"} \oplus \text{"}, \text{"} \otimes \text{"}\rangle,$$
$$\langle \text{"}\boldsymbol{member}\text{"}, \forall_{\langle x\rangle} \underset{C}{\boldsymbol{\epsilon}} [\boldsymbol{x}] \Leftrightarrow \exists_{\langle xr, xi\rangle} \underset{A}{\boldsymbol{\epsilon}} [\boldsymbol{xr}] \wedge \underset{A}{\boldsymbol{\epsilon}} [\boldsymbol{xi}] \wedge$$
$$(\boldsymbol{x} \equiv \langle \boldsymbol{xr}, \boldsymbol{xi}\rangle)\rangle;$$

$$\langle \text{"} \oplus \text{"}, \forall_{\langle x1,\, y1,\, x2,\, y2\rangle} \langle \boldsymbol{x1}, \boldsymbol{y1}\rangle \underset{C}{\oplus} \langle \boldsymbol{x2}, \boldsymbol{y2}\rangle \equiv \langle \boldsymbol{x1} \underset{A}{\oplus} \boldsymbol{x2}, \boldsymbol{y1} \underset{A}{\oplus} \boldsymbol{y2}\rangle\rangle;$$

$$\langle \text{"} \otimes \text{"}, \forall_{\langle x1,\, y1,\, x2,\, y2\rangle} \langle \boldsymbol{x1}, \boldsymbol{y1}\rangle \underset{C}{\otimes} \langle \boldsymbol{x2}, \boldsymbol{y2}\rangle \equiv$$
$$\langle \left(\boldsymbol{x1} \underset{A}{\otimes} \boldsymbol{x2}\right) \underset{A}{\oplus} \underset{A}{\text{inv}} \left[\left(\boldsymbol{y1} \underset{A}{\otimes} \boldsymbol{y2}\right)\right], \left(\boldsymbol{x1} \underset{A}{\otimes} \boldsymbol{y2}\right) \underset{A}{\oplus} \left(\boldsymbol{y1} \underset{A}{\otimes} \boldsymbol{x2}\right)\rangle \rangle;$$
$$\boldsymbol{C}$$
$$];$$

7.2 The Closure Property

$$\text{Closed}[\boldsymbol{A}, \text{"} \oplus \text{"}] \equiv$$
$$\langle\langle \text{"} \oplus .\text{closure"}, \forall_{\langle x,\, y\rangle} \left(\left(\underset{A}{\boldsymbol{\epsilon}} [\boldsymbol{x}] \wedge \underset{A}{\boldsymbol{\epsilon}} [\boldsymbol{y}]\right) \Rightarrow \underset{A}{\boldsymbol{\epsilon}} [\boldsymbol{x} \oplus \boldsymbol{y}]\right)\rangle\rangle;$$

With these notions defined the functor prover in *Theorema* can be called as follows:

$$\text{FunctorProver}[\langle \text{"}\boldsymbol{F2}\text{"}, \Big(\text{Closed}[\boldsymbol{A}, \underset{A}{\oplus}] \wedge$$
$$(\boldsymbol{C} \equiv \text{Complexification}[\boldsymbol{A}])\Big)$$
$$\Rightarrow$$
$$\text{Closed}[\boldsymbol{C}, \underset{C}{\oplus}]\rangle,$$
$$\langle\rangle]$$

We instruct the prover to prove the closure of addition in the domain C given that $Closed[A, \underset{A}{\oplus}] \wedge (C \equiv Complexification[A])$ i.e. addition in A is closed and the domain C is the complexification of A. No additional assumptions are made. The automatically generated proof is given in the appendix. Explanatory text in italics has been included in the proof to show the points at which control is passed from one prover to the other.

8 Conclusion

The system presented in this paper is currently being developed in the frame of the *Theorema* project. Some experiments involving proving in algebraic properties of abstract structures and proving over natural numbers have already been carried out to test the expressiveness of the system. We are also studying how functors can be integrated into the proof search framework we presented. The main research to be carried out is designing appropriate proof control constraints for various fields of mathematics.

References

[Buchberger 1993] B. Buchberger, *Mathematica* : Doing Mathematics by Computer ?, Invited talk at DISCO 93, Gmunden, Austria. Written version in A. Miola, M.Temperini (eds.), Advances in the Design of Symbolic Computation Systems, pp. 1-29, Springer Wien - New York.

[Buchberger 1996] B. Buchberger, Symbolic Computation: Computer Algebra and Logic. In F. Baader, K.U. Schultz eds., Frontiers in Combining Systems, pp. 193-220, Applied Logics Series, Kluwer Academic Publishers, 1996.

[Buchberger et al., 1997] B. Buchberger, T.Jebelean, F. Kriftner, M. Marin, E.Tomuta, D.Vasaru, A Survey of the *Theorema* Project, in: Proceedings of ISSAC'97 (International Symposium on Symbolic and Algebraic Computation), Maui, Hawaii, July 21-23, 1997, W. Kuechlin (ed.).

[Buchberger, Jebelean 1997] B. Buchberger, T.Jebelean The Predicate Logic Prover, Proceedings of the *Theorema* Workshop, June 9-10 1997, Hagenberg, Austria.

[Bundy 1988] A. Bundy The use of explicit plans to guide inductive proofs. In Ewing Lusk and Ross Overbeek, editors, Proceedings of the 9th CADE, p. 111-120, LNCS 310, Springer Verlag Berlin, Germany, 1988.

[Bundy et. al, 1990] A. Bundy, F. van Harmelen, C. Horn, A. Smaill, The Oyster-Clam System. In Mark E.Stickel, editor, Proceedings of the 10th CADE, p. 647-648, LNAI 449, Springer Verlag Berlin, Germany, 1990.

[Gordon et al., 1979] M.J.C. Gordon, A. Milner, C.P. Wadsworth, Edinburgh LCF, LNAI 78, Springer Veralg Berlin Heidelberg new York, 1979.

[Hewitt, 1977] C. Hewitt, Viewing control structures as patterns of passing messages. Artificial Intelligence, Vol. 8 pp. 323-364, 1977.

[Huang et al., 1994] X. Huang, M. Kerber, J. Richts, A. Sehn, Planning Mathematical Proofs with Methods In Journal of Information Processing and Cybernetics, 30: 277-291, 1994

[Fikes, Nilsson, 1970] R.E. Fikes, N.J. Nilsson, STRIPS: a new procedure to the application of theorem proving to problem solving. Artificial Intelligence, 2: 189-208, 1971.

[Wolfram, 1996] S. Wolfram, The *Mathematica* Book, 3rd ed., Wolfram Media/Cambridge University Press, 1996.

Appendix

The predicate logic prover is called.

We prove (F2).

We show

(F2) $\text{Closed}[\boldsymbol{A}, \underset{A}{\oplus}] \wedge (\boldsymbol{C} \equiv \text{Complexification}[\boldsymbol{A}]) \Rightarrow \text{Closed}[\boldsymbol{C}, \underset{C}{\oplus}]$.

We prove (F2) by the deduction rule.

We assume

(F2.H) $\text{Closed}[\boldsymbol{A}, \underset{A}{\oplus}] \wedge (\boldsymbol{C} \equiv \text{Complexification}[\boldsymbol{A}])$ and show

(F2.C) $\text{Closed}[\boldsymbol{C}, \underset{C}{\oplus}]$.

The functor prover inserts one proof step to apply the definition of the "Closed" property.

We expand the definition of

$$\text{Closed}[\boldsymbol{C}, \underset{C}{\oplus}].$$

Thus we have to prove

(F2.C) $\langle\text{"F2.C.C.} \oplus \text{.closure"}, \forall_{\langle x,y\rangle} \underset{C}{\boldsymbol{\epsilon}}[\boldsymbol{x}] \wedge \underset{C}{\boldsymbol{\epsilon}}[\boldsymbol{y}] \Rightarrow \underset{C}{\boldsymbol{\epsilon}}[\big(\boldsymbol{x} \underset{C}{\oplus} \boldsymbol{y}\big)]\rangle$ under the assumptions

(F2.H.1) $\text{Closed}[\boldsymbol{A}, \underset{A}{\oplus}]$,

(F2.H.2) $\boldsymbol{C} \equiv \text{Complexification}[\boldsymbol{A}]$.

Control is returned to the predicate logic prover.

For proving (F2.C.F2.C.C.⊕.closure) we prove, for arbitrary but fixed values,

(F2.C.F2.C.C.⊕.closure') $\underset{C}{\boldsymbol{\epsilon}}[\boldsymbol{x}_0] \wedge \underset{C}{\boldsymbol{\epsilon}}[\boldsymbol{y}_0] \Rightarrow \underset{C}{\boldsymbol{\epsilon}}[\big(\boldsymbol{x}_0 \underset{C}{\oplus} \boldsymbol{y}_0\big)]$.

We prove F2.C.F2.C.C.⊕.closure' by the deduction rule.

We assume

(F2.C.F2.C.C.⊕.closure'.H) $\underset{C}{\boldsymbol{\epsilon}}[\boldsymbol{x}_0] \wedge \underset{C}{\boldsymbol{\epsilon}}[\boldsymbol{y}_0]$

and show

(F2.C.F2.C.C.⊕.closure'.C) $\underset{C}{\boldsymbol{\epsilon}}[\big(\boldsymbol{x}_0 \underset{C}{\oplus} \boldsymbol{y}_0\big)]$.

The goal is atomic with head an operator in the domain C.The functor prover tries to apply the definition of addition in C.

Trying to apply the definition of $\underset{C}{\oplus}$ to

(F2.C.F2.C.C.$\oplus$.closure'.C) $\underset{C}{\boldsymbol{\epsilon}}$ $[\Big(\boldsymbol{x}_0 \underset{C}{\oplus} \boldsymbol{y}_0\Big)]$.

We apply the definition of $\underset{C}{\boldsymbol{\epsilon}}$ to the goal, at position $\langle 2, 1, 1\rangle$

We show
(F2.C.F2.C.C.$\oplus$.closure'.C.C.ϵ) $\underset{C}{\boldsymbol{\epsilon}}$ $[\Big(\langle \boldsymbol{xr}_1, \boldsymbol{xi}_1\rangle \underset{C}{\oplus} \boldsymbol{y}_0\Big)]$

under the assumptions
(F2.C.F2.C.C.$\oplus$.closure'.H.1) $\underset{C}{\boldsymbol{\epsilon}}$ $[\langle \boldsymbol{xr}_1, \boldsymbol{xi}_1\rangle]$,

(F2.C.F2.C.C.$\oplus$.closure'.H.2) $\underset{C}{\boldsymbol{\epsilon}}$ $[\boldsymbol{y}_0]$,

(F2.H.1) Closed $[\boldsymbol{A}, \underset{A}{\oplus}]$,

(F2.H.2) $\boldsymbol{C} \equiv$ Complexification$[\boldsymbol{A}]$,
(F2.C.F2.C.C.$\oplus$.closure'.C.1) $\underset{A}{\boldsymbol{\epsilon}}$ $[\boldsymbol{xr}_1]$,

(F2.C.F2.C.C.$\oplus$.closure'.C.2) $\underset{A}{\boldsymbol{\epsilon}}$ $[\boldsymbol{xi}_1]$.

We apply the definition of $\underset{C}{\boldsymbol{\epsilon}}$ to the goal, at position $\langle 2, 1, 2\rangle$

We show
(F2.C.F2.C.C.$\oplus$.closure'.C.C.ϵ.C.ϵ) $\underset{C}{\boldsymbol{\epsilon}}$ $[\Big(\langle \boldsymbol{xr}_1, \boldsymbol{xi}_1\rangle \underset{C}{\oplus} \langle \boldsymbol{xr}_2, \boldsymbol{xi}_2\rangle\Big)]$

under the assumptions
(F2.C.F2.C.C.$\oplus$.closure'.H.1) $\underset{C}{\boldsymbol{\epsilon}}$ $[\langle \boldsymbol{xr}_1, \boldsymbol{xi}_1\rangle]$,

(F2.C.F2.C.C.$\oplus$.closure'.H.2) $\underset{C}{\boldsymbol{\epsilon}}$ $[\langle \boldsymbol{xr}_2, \boldsymbol{xi}_2\rangle]$,

(F2.H.1) Closed $[\boldsymbol{A}, \underset{A}{\oplus}]$,

(F2.H.2) $\boldsymbol{C} \equiv$ Complexification$[\boldsymbol{A}]$,
(F2.C.F2.C.C.$\oplus$.closure'.C.1) $\underset{A}{\boldsymbol{\epsilon}}$ $[\boldsymbol{xr}_1]$,

(F2.C.F2.C.C.$\oplus$.closure'.C.2) $\underset{A}{\boldsymbol{\epsilon}}$ $[\boldsymbol{xi}_1]$,

(F2.C.F2.C.C.$\oplus$.closure'.C.C.ϵ.1) $\underset{A}{\boldsymbol{\epsilon}}$ $[\boldsymbol{xr}_2]$,

(F2.C.F2.C.C.$\oplus$.closure'.C.C.ϵ.2) $\underset{A}{\boldsymbol{\epsilon}}$ $[\boldsymbol{xi}_2]$.

We apply the definition of $\underset{C}{\oplus}$ to the goal, at position $\langle 2, 1\rangle$

We show
(F2.C.F2.C.C.$\oplus$.closure'.C.C.ϵ.C.ϵ.C.$\oplus$) $\underset{C}{\boldsymbol{\epsilon}}$ $[\langle\Big(\boldsymbol{xr}_1 \underset{A}{\oplus} \boldsymbol{xr}_2\Big), \Big(\boldsymbol{xi}_1 \underset{A}{\oplus} \boldsymbol{xi}_2\Big)\rangle]$

under the assumptions
(F2.C.F2.C.C.$\oplus$.closure'.H.1) $\underset{C}{\boldsymbol{\epsilon}}$ $[\langle \boldsymbol{xr}_1, \boldsymbol{xi}_1\rangle]$,

(F2.C.F2.C.C.$\oplus$.closure'.H.2) $\underset{C}{\boldsymbol{\epsilon}}$ $[\langle \boldsymbol{xr}_2, \boldsymbol{xi}_2\rangle]$, (F2.H.1) Closed $[\boldsymbol{A}, \underset{A}{\oplus}]$,

(F2.H.2) $\boldsymbol{C} \equiv$ Complexification$[\boldsymbol{A}]$,

(F2.C.F2.C.C.$\oplus$.closure'.C.1) $\underset{A}{\boldsymbol{\epsilon}} [\boldsymbol{xr}_1]$,

(F2.C.F2.C.C.$\oplus$.closure'.C.2) $\underset{A}{\boldsymbol{\epsilon}} [\boldsymbol{xi}_1]$,

(F2.C.F2.C.C.$\oplus$.closure'.C.C.ϵ.1) $\underset{A}{\boldsymbol{\epsilon}} [\boldsymbol{xr}_2]$,

(F2.C.F2.C.C.$\oplus$.closure'.C.C.ϵ.2) $\underset{A}{\boldsymbol{\epsilon}} [\boldsymbol{xi}_2]$.

We apply the definition of $\underset{C}{\boldsymbol{\epsilon}}$ to the goal, at position $\langle 2 \rangle$

We show

(F2.C.F2.C.C.$\oplus$.closure'.C.C.ϵ.C.ϵ.C.$\oplus$.C.ϵ) $\underset{A}{\boldsymbol{\epsilon}} [(\boldsymbol{xr}_1 \underset{A}{\oplus} \boldsymbol{xr}_2)] \wedge \underset{A}{\boldsymbol{\epsilon}} [(\boldsymbol{xi}_1 \underset{A}{\oplus} \boldsymbol{xi}_2)]$

under the assumptions

(F2.C.F2.C.C.$\oplus$.closure'.H.1) $\underset{C}{\boldsymbol{\epsilon}} [\langle \boldsymbol{xr}_1, \boldsymbol{xi}_1 \rangle]$,

(F2.C.F2.C.C.$\oplus$.closure'.H.2) $\underset{C}{\boldsymbol{\epsilon}} [\langle \boldsymbol{xr}_2, \boldsymbol{xi}_2 \rangle]$,

(F2.H.1) Closed $[\boldsymbol{A}, \underset{A}{\oplus}]$,

(F2.H.2) $\boldsymbol{C} \equiv$ Complexification$[\boldsymbol{A}]$,

(F2.C.F2.C.C.$\oplus$.closure'.C.1) $\underset{A}{\boldsymbol{\epsilon}} [\boldsymbol{xr}_1]$,

(F2.C.F2.C.C.$\oplus$.closure'.C.2) $\underset{A}{\boldsymbol{\epsilon}} [\boldsymbol{xi}_1]$,

(F2.C.F2.C.C.$\oplus$.closure'.C.C.ϵ.1) $\underset{A}{\boldsymbol{\epsilon}} [\boldsymbol{xr}_2]$,

(F2.C.F2.C.C.$\oplus$.closure'.C.C.ϵ.2) $\underset{A}{\boldsymbol{\epsilon}} [\boldsymbol{xi}_2]$.

Control is returned to the predicate logic prover which completes the proof.

We prove the individual conjunctive parts of (F2.C.F2.C.C.$\oplus$.closure'.C.C.ϵ.Cϵ.C.$\oplus$.C.ϵ):

Proof of (F2.C.F2.C.C.$\oplus$.closure'.C.C.ϵ.C.ϵ.C.$\oplus$.C.ϵ.1)

$\underset{A}{\boldsymbol{\epsilon}} [(\boldsymbol{xr}_1 \underset{A}{\oplus} \boldsymbol{xr}_2)]$:

From (A.$\oplus$.closure) by specialization we obtain

(A.$\oplus$.closure/3) $\underset{A}{\boldsymbol{\epsilon}} [\boldsymbol{xr}_1] \wedge \underset{A}{\boldsymbol{\epsilon}} [\boldsymbol{xr}_2] \Rightarrow \underset{A}{\boldsymbol{\epsilon}} [(\boldsymbol{xr}_1 \underset{A}{\oplus} \boldsymbol{xr}_2)]$.

For proving (F2.C.F2.C.C.$\oplus$.closure'.C.C.ϵ.C.ϵ.C.$\oplus$.C.ϵ.1), by (A.$\oplus$.closure/3), it suffices to prove

(A.$\oplus$.closure/3.A) $\underset{A}{\boldsymbol{\epsilon}} [\boldsymbol{xr}_1] \wedge \underset{A}{\boldsymbol{\epsilon}} [\boldsymbol{xr}_2]$.

We prove the individual conjunctive parts of (A.$\oplus$.closure/3.A):

Proof of (A.$\oplus$.closure/3.A.1) $\underset{A}{\boldsymbol{\epsilon}} [\boldsymbol{xr}_1]$:

Formula (A.$\oplus$.closure/3.A.1) is true because it is identical to (F2.C.F2.C.C.$\oplus$.closure'.C.1).

Proof of (A.$\oplus$.closure/3.A.2) $\underset{A}{\boldsymbol{\epsilon}} [\boldsymbol{xr}_2]$:

Formula (A.$\oplus$.closure/3.A.2) is true because it is identical to (F2.C.F2.C.C.$\oplus$.closure'.C.C.ϵ.1).
Proof of (F2.C.F2.C.C.$\oplus$.closure'.C.C.ϵ.C.ϵ.C.$\oplus$.C.ϵ.2) $\underset{A}{\boldsymbol{\epsilon}}\,[\Big(\boldsymbol{xi}_1 \underset{A}{\oplus} \boldsymbol{xi}_2\Big)]$:

From (A.$\oplus$.closure) by specialization we obtain
(A.$\oplus$.closure/3) $\underset{A}{\boldsymbol{\epsilon}}\,[\boldsymbol{xi}_1] \wedge \underset{A}{\boldsymbol{\epsilon}}\,[\boldsymbol{xi}_2] \Rightarrow \underset{A}{\boldsymbol{\epsilon}}\,[\Big(\boldsymbol{xi}_1 \underset{A}{\oplus} \boldsymbol{xi}_2\Big)]$.

For proving (F2.C.F2.C.C.$\oplus$.closure'.C.C.ϵ.C.ϵ.C.$\oplus$.C.ϵ.2), by (A.$\oplus$.closure/3), it suffices to prove
(A.$\oplus$.closure/3.A) $\underset{A}{\boldsymbol{\epsilon}}\,[\boldsymbol{xi}_1] \wedge \underset{A}{\boldsymbol{\epsilon}}\,[\boldsymbol{xi}_2]$.

We prove the individual conjunctive parts of (A.$\oplus$.closure/3.A):
Proof of (A.$\oplus$.closure/3.A.1) $\underset{A}{\boldsymbol{\epsilon}}\,[\boldsymbol{xi}_1]$:

Formula (A.$\oplus$.closure/3.A.1) is true because it is identical to (F2.C.F2.C.C.$\oplus$.closure'.C.2).Proof of (A.$\oplus$.closure/3.A.2) $\underset{A}{\boldsymbol{\epsilon}}\,[\boldsymbol{xi}_2]$:

Formula (A.$\oplus$.closure/3.A.2) is true because it is identical to (F2.C.F2.C.C.$\oplus$.closure'.C.C.ϵ.2).