Reasoning on Data-Parallel Programs in Isabelle/HOL

Matthias Daum

Saarland University,
Saarbrücken, Germany
<md11@wjpserver.cs.uni-sb.de>

Abstract This paper describes a dialect of the programming language C for data-parallel applications and its embedding into a verification environment based on Isabelle/HOL. On the one hand, the proposed language eliminates C features that interfere with program verification, while on the other hand, it introduces means to express data-parallelism. The embedding of this language into the existing verification environment shows the flexibility and diversified usability of the used verification environment even for rather special-purpose programming languages.

1 Introduction

Data-Parallel C Extensions (DPCE) [1] define a set of extensions to the programming language C [2] that support the programming of data-parallel applications. The so-extended programming language permits the implementation of concise and portable programs that will efficiently run on data-parallel architectures. DPCE comprise the most fundamental concepts of a consistent data-parallel model. The language adopts a general model that includes the basic concept of parallel data aggregates with an inner structure, a nodal distribution, and a context (constraining the participating elements).

While the extended language permits convenient programming, we encounter obviously the same problems on verification as with C. In order to reduce verification costs, I have restricted this language to a type-safe subset called *Data-Parallel C0*. For verification, I specified a formal semantics for Data-Parallel C0 (DPC0). This semantics employs Schirmer's verification environment for imperative programming languages [3] as its basis.

The verification environment is implemented on top of the generic theorem-prover Isabelle [4,5]. It features a decent abstract syntax permitting to embed a great range of sequential imperative programming languages. Schirmer proposed an embedding for a type-safe C variant. The DPC0 embedding reuses great portions of this work and mainly adds data-parallel specific language features. I have implemented a syntax-translation tool in order to rule out any accidental mistakes during translation.

The remaining paper is organized as follows: The basic language features of DPC0 are described in the next section. Section 3 introduces Schirmer's verification environment. In Section 4, the formal semantics is defined. And finally, Section 5 concludes the paper, gives an outlook on further work, and points to related work.

2 Data-Parallel C0

The programming language Data-Parallel C0 (DPC0) was developed as a variant of Data-Parallel C (DPC) that is just as well suited for verification as for programming of data-parallel applications. The most notable difference is the perfect type safety of DPC0 because type safety tremendously simplifies the verification effort for individual programs. For the same reason, expressions have to be free of side effects.

Other crucial design objectives were compatibility and easy portability of existing code. These objectives do not only help a programmer when switching; they were most fundamental because I did not intend to write a new compiler. Instead, the source code has to be compiled with a standard DPC compiler. Hence, DPC0 is almost literally compatible to DPC. The only syntactic difference regards dynamic memory allocation.

Moreover, DPC0 inherited the basic design principles of DPC: The ability to write efficient programs for data-parallel systems in a readable and portable manner. Hence, both languages feature a very general model for parallel data. The key concept is the separation of algorithm and data distribution. The programmer can naturally specify the data distribution in the high level language but will then program independently from it. On the semantic level, there is just a single thread of control.

The remaining section sketches the key concept of data-parallelism in both languages, focusing on DPC0 but pointing out important differences to DPC.

A data-parallel object is an aggregate of data elements of the same data type. The data elements might be distributed over several processors or *nodes*. Moreover, they have an inner structure, i. e., a rank and dimensions. Hence, a data aggregate is characterized on the one hand by the data type of the elements, and on the other hand by the inner structure and the nodal distribution. The latter two attributes together form a *shape*. Here is a small example definition of a shape for illustration:

```
shape [2][5][3] S;
```

This line defines a new shape called `S` with rank three. The first dimension has two positions, the second one five, and the third one three. The nodal distribution was not explicitly specified and can thus be chosen by the compiler.

Additionally, a *context* is attached to each shape. The context limits the scope of operations only to the so-called *active positions* in the aggregate. While rank,

dimensions, and distribution are static properties, the context might dynamically change during the program execution. Hence, it will never be part of a definition but can be altered with special commands, namely `where` and `everywhere`.

The syntax of the where statement reads as follows:

`where (`⟨*mask*⟩`)` ⟨*statement block*⟩

We assume that ⟨*mask*⟩ is a parallel Boolean expression of some shape `S`. In the specified ⟨*statement block*⟩, all those positions of `S` are deactivated, where the element in ⟨*mask*⟩ evaluates to `false`. [1] The where statement might optionally have an `else` branch with a second statement block. During the execution of this block, all those positions are deactivated, where the elements of the mask evaluate to `true`.

DPC features the statement everywhere that will temporarily reactivate all positions of a shape for a statement block. There was no use-case for this statement in our project and hence it is currently not supported in DPC0. However, it could easily be added. Moreover, the same effect can be enforced by a function call, which will implicitly activate all positions.

A parallel data type is fully described by its element type and its shape. Type and shape are separated by colons, as illustrated in the following example:

```
int:S parallel;
```

This line defines a parallel variable that contains integer elements and is shaped according to shape `S`.

Operations on data-parallel objects comprise the usual C operations like arithmetics, logical operations and assignments. Unary operations work just element-wise on all active positions. So do binary operations, given two compatible parallel expressions. If a binary operator is applied to a scalar and a parallel expression, DPC will perform an implicit conversion. In contrast, DPC0 does not support any implicit conversions. However, both language variants feature a type-conversion syntax to construct parallel values from scalars, called *replication* in data-parallel programming.

Moreover, both languages introduce a number of novel operators. Those include reduction operators, a minimum and a maximum operator together with their respective reduction operators as well as so-called *parallel indexing*.

The latter operation permits on the one hand the selection of a single element just as known from arrays. On the other hand, a parallel value may arbitrarily be reshaped with parallel indexing – a technique that is called *scatter and gather* in data-parallel programming. DPC permits even a third kind of parallel indexing, which is called sliced indexing. However, this operation is not fully type safe and thus not supported in DPC0.

[1] Strictly speaking, there is no value `false` in C and hence in DPC. Instead, value 0 is used to represent the logical value false. The standard library header `<stdbool.h>` defines the necessary macros in order to overcome this shortcoming.

The common syntax for both in DPC0 supported forms reads as follows:

`[`$\langle i_0 \rangle$`]` ... `[`$\langle i_n \rangle$`]` $\langle pexpr \rangle$

We assume that $\langle pexpr \rangle$ is a parallel expression. The expression is only valid if the number of index expressions is equal to the rank of the parallel expression's shape. The index expressions $\langle i_j \rangle$ can either be scalar or parallel integral values. In the first case, the expression evaluates to the selected scalar. In the second case, all index expressions have to be of the same shape; a new parallel value is constructed with that shape.

The new value might as well be used as left-value, where the evaluation order effects the result if a position is selected multiple times, e. g.:

```
shape [5] S;  long:S p,q;

[(int:S) 1] p = q;
```

The new value of variable `p` is implementation defined in DPC, while DPC0 treats this case as a runtime error.

In analogy to the original syntax of the language, we will denote parallel indexing in the paper by $[i_0] \ldots [i_{rankof(S)-1}]x$ for some parallel expression x with shape S.

DPC inherited the weak type system from C and additionally permits partially and even completely unspecified shapes – in particular the generic shape `void` –, which are an offence against type safety. However, the specification of DPC includes several library functions that rely on the weak type system. Some of these functions had to be dropped in DPC0, for instance the function `newshape` that can dynamically create a new shape.

Other library functions were replaced by additional operators. In DPC, dynamic memory allocation for non-parallel data objects works C-like with the library function `malloc` and likewise for parallel data objects with the additional library function `palloc`. The counterparts for memory allocation in DPC0 are the built-in operators `new` and `pnew`, which return a typed pointer in a C++-like fashion.

Another newly introduced operator is called `pcoord`. The corresponding library function in DPC designates a parallel value that enumerates the elements along an axis, formally:

Assuming S is a shape and a denotes an axis, i. e., $a < rankof(S)$, we define $pcoord(S, a) \equiv P$ such that P is the parallel value of shape S with the element type `int` and $\forall i_0, \ldots, i_{rankof(S)-1} : [i_0] \ldots [i_{rankof(S)-1}]P = i_a$.

This functionality is frequently used for indexing.

In order to contain the number of newly introduced operators in DPC0, functions might still take an arbitrary shape as argument. DPC0 treates shapes as values of a distinct data type `shape`. A newly defined shape is a constant of that type. It can be used in the specification of a parallel data type and equally

as an argument to a function. With the parameter of this function, however, it is impossible to specify a new parallel data type because that is not a constant anymore. However, this approach permits the access of shape properties like its rank (`rankof`) or its elements along an axis (`dimof`).

3 A Verification Environment for Sequential Imperative Programming Languages

This section shortly introduces the very basics of Schirmer's verification environment [3]. This environment is seamlessly integrated into the generic proof assistant Isabelle [4] using the higher order logic library HOL [5]. The composed system provides a sound and practically usable environment for reasoning on sequential imperative programs.

The verification environment is based on a general language model for sequential imperative programs, which is called *Simpl*. This model is not restricted to a particular programming language but covers all common language features of imperative programs: mutually recursive functions, global and stack variables, pointers, heap objects and dynamic memory allocation, runtime faults like array bound violations or dereferencing null pointers, and much more.

The flexibility of Simpl is based on the mixed embedding scheme for statements and expressions: While statements are represented by an abstract data type, the expressions in the language model are represented by ordinary Isabelle/HOL expressions. For this language model, Schirmer has defined an operational semantics and a Hoare logic for both, partial and total correctness. Furthermore, the Hoare logics were proven sound and complete with respect to the operational semantics. Formalisation and proofs were completely conducted in the theorem prover Isabelle/HOL. A *verification condition generator* (VCG) translates the deep-embedded statements into Isabelle/HOL formulæ. The VCG is integrated in the proof assistant as a proof tactic.

The remaining section sketches the representation of program states and describes the syntax of the language in more detail.

Program States. In his language model, Schirmer does not fix a particular representation of the program state space. However, he proposes the use of records as provided by Isabelle/HOL. Records are essentially tuples but supplemented with selection and update functions for each component. For example,

$$\textbf{record}\ \mathit{state} = N :: \mathit{nat} \quad B :: \mathit{bool}$$

defines a record type that is represented as a tuple of a natural number and a Boolean value. Additionally, this definition creates two selectors $N \in \mathit{state} \to \mathit{nat}$ and $B \in \mathit{state} \to \mathit{bool}$ as well as two update functions $N\text{-}\mathit{update} \in \mathit{nat} \to \mathit{state} \to \mathit{state}$ and $B\text{-}\mathit{update} \in \mathit{bool} \to \mathit{state} \to \mathit{state}$.

A record update $N\text{-}\mathit{update}\ n\ s$ can be abbreviated with $s(\!|N := n|\!)$ in Isabelle/HOL.

We represent each program variable by a record field. Thus, every variable might have an individual HOL type and the automatic type inference of Isabelle/HOL takes care of typing issues. Moreover, with field selection and update we have convenient means to express state updates and assertions.

Syntax of the Programming Language. The abstract statement syntax is defined by the polymorphic data type $(\mathcal{S}, \mathcal{P}, \mathcal{F})$ *com* over a program state space $\mathcal{S}$, a procedure name space $\mathcal{P}$, and a fault type $\mathcal{F}$. Table 1 lists the constructors of the data type that are used with DPC0.[2] The variables c, c_1 and c_2 are Simpl commands of type $(\mathcal{S}, \mathcal{P}, \mathcal{F})$ *com*. The Boolean condition b and the guard g are modelled as state sets of type $\mathcal{S}$ *set*.

Table 1. Statement syntax

Atomic steps	
Skip	Do nothing
Basic f	Basic command, where $f \in \mathcal{S} \to \mathcal{S}$ is a state update
Standard control flow	
Seq c_1 c_2	Sequential composition
Cond b c_1 c_2	Conditional execution (if-then-else)
While b c	Loop
Runtime faults	
Guard f g c	Guarded command, where $f \in \mathcal{F}$ is a fault flag
Scoping and procedure calls	
Block init c exit	Statement block with local variables; *init* $\in \mathcal{S} \to \mathcal{S}$ initializes, *exit* $\in \mathcal{S} \to \mathcal{S} \to \mathcal{S}$ restores variables
Call init p exit	Call to a procedure, where $p \in \mathcal{P}$ is the procedure name, *init* $\in \mathcal{S} \to \mathcal{S}$ and *exit* $\in \mathcal{S} \to \mathcal{S} \to \mathcal{S}$ pass parameters resp. result

The semantic rules for the common language constructs follow the standard textbook semantics. The command *Basic f* applies the function f to the current state. Examples for basic commands are assignments or dynamic memory allocation. As conditions are semantically modeled as state sets, testing the conditions is formulated as set membership.

During expression evaluation, we might encounter runtime faults like division by zero or the dereferencing of a `NULL` pointer. We use the guarded command *Guard f g c* to model these faults. In order to prove a guarded command, we have to ensure that the guard g holds. The fault flag f is used for the integration with automatic tools and not of interest throughout the scope of this paper.

The abstract syntax differs much from the original appearance in an ordinary imperative programming language. Hence, Schirmer has employed the powerful

[2] Schirmer's data-type definition is somewhat more general because he deals with language features like abrupt termination and dynamic method invocation. As these features are not present in DPC0, we may safely simplify matters, here.

syntax-translation mechanisms of Isabelle in order to establish a more readable, so-called *concrete syntax*.

So, for example, an assignment of variable b to variable a looks like

$$'a :== 'b$$

while Isabelle/HOL internally processes it as:

$$Basic\ (\lambda s.\ s(\!|a := b\ s|\!))$$

4 Embedding DPC0 Code in Isabelle/HOL

In this section, we explore the semantics of DPC0. We regard usual language concepts like type system, variables, operations, and statements. A few peculiarities of the language design are explained, here, because they can best be understood together with the semantical background.

Type system. Type `bool` is naturally represented as type *bool* in Isabelle/HOL. All other unsigned integral types are represented as type *nat* and all signed integral types as type *int*. The respective type ranges are enforced separately via guards.

Array and parallel types are represented as lists. Note that multi-dimensional arrays in C are arrays with an elementary type of an array. Consequently, they are represented as lists of lists. Parallel types, however, are represented as lists of their elementary type regardless of the inner structure. This approach is motivated by the observation that the inner structure does not matter for most parallel operations. Wherever the inner structure is relevant, the syntax-translation tool explicitly provides it as additional parameter to the operator.

As mentioned in Section 2, DPC0 integrates the notion of shapes as one distinct type into the type system. However, the concept of a shape is actually twofold. On the one hand, a shape definition contains static properties, namely rank and dimensions, and optionally a nodal distribution. As the semantics does not depend on the nodal distribution, we can completely disregard it. Rank and dimensions, we preserve as a constant of type *nat list*. On the other hand, a context is set up for each shape, and this context might change during program execution. Consequently, it has to be part of the program state just like an ordinary variable. We represent contexts as components of the state record of type *bool list*.

Hence, the shape definition

```
shape [2][5][3] S;
```

is represented in Isabelle/HOL as constant *s'S* and record component *c'S*:

constdefs $s'S \equiv [2, 5, 3]$
record $state = \ \ldots\ \ c'S :: bool\ list\ \ \ldots$

Variables and the address-of operator. Usually, variables are components of the state record just as said in Section 3. However, there is one exception to that rule: Like C, DPC0 features an address-of operator that constructs a pointer to a variable. Schirmer did not provide a semantics for such a construct because it leads to aliasing of variables, which cause a considerable overhead in verification. However, he already takes care of aliasing for pointers to dynamically allocated heap objects. Moreover, we can statically check for a given program, where the address-of operator is applied to. Hence, we introduce the notion of *addressable variables* and store them on the heap. Note that only global variables may be addressable in DPC0 because otherwise, we might introduce dangling pointers.

The very same feature is also used for the call-by-reference semantics of array arguments. We just regard arrays as addressable if they are used as an argument in a function call. The drawback is that hence, arrays must be global if passed as argument to a function.

Variable initialization. Our representation of expressions requires that all memory objects have to be initialized prior to their first use. The undetected use of an uninitialized pointer, for example, would completely destroy type safety. Hence, we must either detect such a case within Isabelle/HOL and guard against it or ensure initialization externally. An explicit modelling of uninitialized values, however, would cause an overhead in verification. So, we take the second option.

C initializes all global variables right upon program start. In contrast, stack variables and dynamic memory objects might be used uninitialized. Here, the programmer – or in our case the verification engineer – has to ensure that initialization takes place, properly. For stack variables, we perform a static definite-assignment analysis. This analysis is a safe over-approximation and is implemented in the syntax-translation tool. A caveat on the static analysis is that it cannot recognize an iterative array initialization in a loop. Thus, stack array variables have to be initialized at once with an explicit initializer although, in a particular case, this might lead to less efficiency. The initializers are converted into a conventional assignment.

A comparable check for dynamic memory objects is naturally impossible. Here, we postulate an implicit default-initialization and have to bear possible extra costs. We can enforce the initialization upon dynamic memory allocation because this mechanism is DPC0-specific, anyway. The syntax for memory allocation was specifically designed such that it can easily be wrapped to the usual DPC mechanisms via the C preprocessor.

Data-Parallel Operations. The semantics of all element-wise working operators simply employs the *map* operator that is build into Isabelle/HOL for element-wise operations on lists. As we have learned in Section 2, the context limits the scope of operations to the active positions. However, for element-wise operations it does actually not matter whether the values for the inactive elements are computed as long as runtime errors are checked context-sensitively.

Reduction operators have to filter the active positions according to the current context; then we can use the built-in list reduction *foldl*.

The inner structure of a parallel expression is only relevant for parallel indexing. We define an auxiliary function *ppos* that computes the position in the flattened list according to some shape for a given list of index expressions. Formally:

$$ppos\ S\ I \equiv \sum_{i=0}^{length(I)-1} I\,!\,i \cdot \prod_{j=i+1}^{length(S)-1} S\,!\,j$$

where ! denotes Isabelle/HOL's operator to select an element from a list. For scalar index expressions, we can directly employ ! on the parallel expression, once that we have computed the position in the flattened list. With a few more standard list manipulations, we achieve the same for parallel index expressions.

Runtime faults. In general, we encounter the same runtime faults in DPC0 as Schirmer does in his C dialect: array bound violations, arithmetic over- and underflows, division by 0, and dereferencing of `NULL` pointers. Additionally, we have to guard against another runtime error: Parallel indexing might be used in assignments, but the specification of DPC does not define an order, in which assignments have to be performed. In order to avoid non-determinism, DPC0 treats it as an error if a single position is selected multiple times.

Unfortunately, we cannot fully reuse the guard generating mechanism of the verification environment because it assumes that each C type is represented by a distinct Isabelle/HOL type. Thus, I decided to generate the guards outside by the syntax-translation tool.

Statements. DPC0 features the same statements as C, hence we can directly adopt the existing framework as is. However, there is one additional statement, `where`, that constraints the context of a shape for the execution of a given statement block.

As we know, a context is represented as pseudo variable. When constraining it for the duration of a statement block, we have to retain the old value of the context and restore it afterwards. Hence, a where statement is equivalent to a statement block with a block-local context variable that is initialized with the constrained value. Assuming the following concrete syntax of where statements

$$\textsc{where}\ \langle mask\rangle\ \textsc{for}\ '\langle context\rangle\ \textsc{do}\ \langle statements\rangle\ \textsc{erehw}$$

we can easily express the semantics by

$$\begin{aligned} &Block\ (\lambda s.\ s(\!|\langle context\rangle := \langle context\rangle\ s \wedge_{\mathrm{p}} \langle mask\rangle|\!)) \\ &\quad \langle statements\rangle \\ &\quad (\lambda s\ t.\ t(\!|\langle context\rangle := \langle context\rangle\ s|\!)) \end{aligned}$$

where operator $\wedge_{\mathrm{p}}$ performs an element-wise Boolean and. If a where statement features an else branch, its semantics is analogously a sequence of two similar *Block* statements.

The current context is relevant for assignments. In the simplest case, there is an assignment to a plain variable within an unconstrained context. Only in this case, we can completely replace the contents of the variable. Otherwise we have to use a selective update, which naturally complicates matters for verification. Happily, we can statically check whether the context is constrained because a function call implicitly activates all positions. The syntax-translation tool distinguishes these cases and supplies the current context to the assignment syntax only when needed.

Figure 1 shows a code fragment together with its translation and illustrates both cases in contrast. The displayed function *elem2par* replicates a scalar value thus constructing a list that represents a parallel value with the given shape.

```
i = (int:S) 0;                 ´i :== elem2par s'S 0;

where (j > (int:S) 3)          WHERE ´j >p elem2par s'S 3
{                              FOR ´c'S DO
  i = j - (int:S) 3;             IN ´c'S:  ´i :== ´j − elem2par s'S 3
}                              EREHW
```

Figure 1. Translation of assignments: unconstrained vs. constrained context

5 Conclusion

Summarizing, this paper has presented a formal semantics of a data-parallel language. The embedding of even this rather special-purpose programming language was possible with few extensions of the well-established verification environment of Schirmer. Most importantly, these extensions provide only semantics for additional expressions and some supplemental syntactic sugar. It was not necessary to touch the kernel of the verification environment, which consists of the abstract syntax and the operational semantics of statements. Hence, this work demonstrates the flexibility and diversified usability of the used verification environment.

The herein presented work comprises on the one hand a syntax-translation tool and on the other hand Isabelle/HOL theories defining the necessary extensions. The tool translates DPC0 source code into its Isabelle/HOL representation and additionally performs several checks ensuring (a) well-typedness, (b) definedness of all variables, type names, functions and struct components, (c) correctness of all function calls with respect to the number and types of their arguments, as well as (d) initialization of all stack variables before their first use. Its implementation consists of about 5 000 lines of code.

The Isabelle/HOL theories define the semantic extensions for expressions as well as some additional concrete syntax for the where statement and the

expressions. Moreover, I have proven a few simple lemmata on the semantic extensions. The library of these lemmata is currently quite small and should be expanded in the future. Currently, the theories count around 500 lines.

As first case studies, I have conducted proofs of the functional correctness for a few rather simple programs. For these small programs, verification could be done easily and quite straightforward. The verification of a larger and more challenging program has yet to be done.

Further Work. Many additional features are desirable in the future: Most notably, data-parallel programming is usually employed in a context of extensive numeric computation. In this field, it is hard to find applications that will not heavily use IEEE-compatible floating-point arithmetic, which is currently not supported by DPC0. Harrison [6,7] works since several years on formal verification of floating-point algorithms using the HOL Light theorem prover. Especially exciting is that it might even be possible to import his theory files directly into Isabelle/HOL because there exists a converter.

Moreover, it turned out that perfect type safety immensely contradicts a 100% compatibility with the C language. It might be worthwhile to investigate an alternative, untyped memory model for Simpl. Tuch and Klein [8], for instance, have developed an alternative heap model for better C compliance. This model does not rely on a well-typed memory but still offers a typed view of memory where possible.

Related Work. Gordon [9] has first presented the idea to embed a programming language into higher order logic. He introduced a while language with variables ranging over natural numbers.

There exists a great range of formal semantics for C dialects. Gurevich and Huggins [10] have presented a formal operational semantics for C based on evolving algebras. However, this semantics was only formalized on paper and not mechanically checked. Tews *et al* [11,12] developed a denotational semantics for a subset of C++ using coalgebras and proposed an embedding in the theorem prover PVS. They used a shallow embedding for both, statements and expressions. The proposed embedding features many advanced language features like arbitrary jumps or side-effects in expressions. While it is impressing to see the formalization of these rather difficult phenomena, the verification of even simple programs suffers from the complex model. The same problem is exposed by Norrish's formalization of the C language [13] that he carried out in the HOL theorem prover.

Even Schirmer provides a fully deep embedded semantics of his restricted C dialect in his work [3]. This semantics allows him to perform definite assignment and well-typedness checks directly in Isabelle/HOL. A simulation relation links the deep embedding with Simpl and is used to transfer results. This approach is very valuable in the context of pervasive verification and in conjunction with a semantical stack for the embedding of inline assembler code. However, it would have been overkill in our subproject.

To the best of my knowledge, it exists no other embedding of a data-parallel language into a theorem prover. Prensa [14] presented a verification environment for fully parallel programs. However, she deals with true parallelism while DPC0 has just a single thread of control on the semantic level. Much more importantly, Prensa's verification enironment lacks procedures.

References

1. Stanberry, L., ed.: Data Parallel C Extensions. Numerical C Extensions Group of X3J11, DPCE Subcommittee, Livermore, CA 94551 (1994)
2. American National Standards Institute: ANSI/ISO/IEC 9899-1999: Programming Languages — C. American National Standards Institute, New York, USA (1999)
3. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, TU Munich (2006)
4. Paulson, L.C.: Isabelle – A Generic Theorem Prover. Volume 828 of Lecture Notes in Computer Science. Springer (1994)
5. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Volume 2283 of Lecture Notes in Computer Science. Springer (2002)
6. Harrison, J.: Theorem Proving with the Real Numbers. Springer (1998)
7. Harrison, J.: Floating-point verification using theorem proving. In Bernardo, M., Cimatti, A., eds.: SFM-6. Volume 3965 of Lecture Notes in Computer Science., Bertinoro, Italy, Springer (2006) 211–242
8. Tuch, H., Klein, G.: A unified memory model for pointers. In Sutcliffe, G., Voronkov, A., eds.: LPAR-12. Volume 3835 of Lecture Notes in Computer Science., Jamaica (2005) 474–488
9. Gordon, M.J.C.: Mechanizing programming logics in higher-order logic. In Birtwistle, G.M., Subrahmanyam, P.A., eds.: Current Trends in Hardware Verification and Automatic Theorem Proving, Banff, Canada, Springer (1988) 387–439
10. Gurevich, Y., Huggins, J.K.: The Semantics of the C Programming Language. In Börger, E., Kleine Büning, H., Jäger, G., Martini, S., Richter, M.M., eds.: Computer Science Logic. Volume 702. Springer (1993) 274–309
11. Hohmuth, M., Tews, H.: The semantics of C++ data types: Towards verifying low-level system components. In: TPHOLs, Emerging Trends, Rome, Italy (2003)
12. Hohmuth, M., Tews, H., Stephens, S.G.: Applying source-code verification to a microkernel – the VFiasco project. In: Tenth ACM SIGOPS European Workshop. (2002)
13. Norrish, M.: C formalised in HOL. PhD thesis, University of Cambridge (1998)
14. Prensa Nieto, L.: Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL. PhD thesis, TU Munich (2002)